DOCUMENT RESUME

ED 089 747 IR 000 459

AUTHOR	Lukas, George; Feurzeig, Wallace
TITLE	Technology for Analysis of Student Interactions With Complex Programs. Final Report for Period January 1972-February 1973.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	BOLTBER-74-2625
PUB DATE	Feb 73
NOTE	221p.
EDRS PRICE	MF-$0.75 HC-$10.20 PLUS POSTAGE
DESCRIPTORS	Algorithms; *College Students; Computer Assisted Instruction; *Computer Programs; Computer Science; *Computer Science Education; *Educational Diagnosis; Higher Education; Man Machine Systems; Problem Solving; Program Descriptions; Program Evaluation; *Programing; Programing Languages
IDENTIFIERS	*Dribble File; Heuristic Methods; LOGO

ABSTRACT

A description is provided of a computer system designed to aid in the analysis of student programing work. The first section of the report consists of an overview and user's guide. In it, the system input is described in terms of a "dribble file" which records all student inputs generated; also an introduction is given to the aids developed for monitoring and analyzing student programing activities. The next section offers a detailed description of the system, including full program documentation, while the final two parts deal with the standard analysis packages developed to facilitate applications and with examples of system use. Details are provided on the manner in which users can scan structures derived from the "dribble files", choosing data of interest, form of presentation, and level of interpretation and moving across these freely in time. General facilities for developing new analysis procedures are described and a technical description of the system's LOGO language is appended. (Author)

BOLT BERANEK AND NEWMAN INC

CONSULTING · DEVELOPMENT · RESEARCH

BOLTBER-74-2625

TECHNOLOGY FOR ANALYSIS OF STUDENT INTERACTIONS WITH COMPLEX PROGRAMS

George Lukas
Wallace Feurzeig
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

U S DEPARTMENT OF HEALTH
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY.

February 1973

Final Report for Period January 1972 - February 1973

Prepared for
NATIONAL SCIENCE FOUNDATION
Education Directorate
Office of Experimental Projects and Programs
5225 Wisconsin Ave., N.W.
Washington, D. C. 20550

Contract NSF-C 708

CAMBRIDGE WASHINGTON, D.C. CHICAGO HOUSTON LOS ANGELES SAN FRANCISCO

BIBLIOGRAPHIC DATA SHEET

1. Report No.

2.

3. Recipient's Accession No.

4. Title and Subtitle
Technology for Analysis of Student Interactions With Complex Programs

5. Report Date
February 1973

6.

7. Author(s) George Lukas and Wallace Feurzeig

8. Performing Organization Rept. No. 2625

9. Performing Organization Name and Address

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NSF-C 708

12. Sponsoring Organization Name and Address

National Science Foundation
5225 Wisconsin Ave., N.W.
Washington, D. C. 20550

13. Type of Report & Period Covered
Final
Jan. 1972-Feb. 1973

14.

15. Supplementary Notes

16. Abstracts
A computer system to aid in the analysis of student programming work is described. The input to this system is a "dribble file" recording all student inputs generated during a student computer interaction. The system provides teachers and researchers a set of aids for monitoring and analyzing the student's programming activity. The user can design the particular form of analysis he desires. During the analysis he can scan structures derived from the dribble files dynamically, choosing data of interest, form of presentation, and level of interpretation, and moving across these freely in time. The design and implementation of this analysis system are described. Its standard mode of use is illustrated and special analysis packages are developed. General facilities for developing new analysis procedures are described. Examples show the application of these various capabilities.

17. Key Words and Document Analysis. 17a. Descriptors

Computer Programming
Education
Psychology
Problem Solving
Heuristic Methods
Reasoning
Algorithms
Diagnostic Routines
Monitor Routines

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group 05-08 05-09 09-02

18. Availability Statement
Release unlimited

19. Security Class (This Report)
UNCLASSIFIED

20. Security Class (This Page
UNCLASSIFIED

21. No. of Pages

22. Price

TABLE OF CONTENTS

Page
ABSTRACT ii
INTRODUCTION iii

Part 1. User's Guide

1. Programming Languages in Teaching 1-1
2. An Example of Use of the LOGO Language 1-2
3. Dribble Files 1-8
4. Building a Language and Monitor System for Processing Dribble Files 1-13
5. The Raw Dribble File 1-22
6. Preprocessing of the Raw Dribble File 1-25
7. Parsing of the Dribble File 1-28
8. An Example of the Output of Parsing 1-35
9. RUNning the Dribble File 1-38

Part 2. System Documentation

1. Introduction to System Documentation 2-1
2. The Use of LOGO as Both System and Object Language . . 2-2
3. The Preprocessing 2-4
4. Parsing 2-7
5. Running 2-20
6. The Display Facility 2-35

Part 3. Analysis Package

1. Introduction to Analysis Package 3-1
2. User Definition of Analysis Procedures 3-1
3. User Augmentation of the Parsing Procedures 3-4
4. Aids for Execution and Debugging of Student Procedures 3-8

Part 4. Examples of System Use

1. Introduction 4-1
2. Example 1 4-2
3. Example 2 4-14
4. Example 3 4-30
5. Example 4 4-36
APPENDIX A

STUDENT INTERACTIONS WITH COMPLEX PROGRAMS

ABSTRACT

The individualization in classroom work made possible by student use of high level programming languages creates new educational and technological challenges. Unless severe time or usage constraints are imposed on their mode of work, the quantity of information generated by the students is much too large for either a teacher or researcher to handle effectively. The computer itself, however, can be used to great advantage for those forms of analysis of student programming work which do not totally depend on problem content. "Dribble files" can be generated containing all the student-computer interaction, and a computer system developed to help teachers or researchers monitor and analyze these dribble files.

A facility of this kind has been developed. It provides suitable primitives and procedure-writing capabilities to enable the user to design the exact form of analysis he desires. During the analysis the user can scan forms derived from the dribble files dynamically, choosing the data of interest, form of presentation, and level of interpretation, and moving across these freely in time.

The design and implementation of this analysis system are described. Its standard mode of use is illustrated and special analysis packages are developed. General facilities that permit users to develop their own analysis procedures are described. Examples are given to illustrate application of these various capabilities.

INTRODUCTION

Programming languages provide contexts within which a great variety of formal processes can be concretely represented and performed. They permit a great scope for individuality of student work both in the number of things students can do, as well as the number of different ways they can do them. Thus, at least in principle, it is reasonable to expect such languages to play an increasingly important role in teaching. This same variety and individuality, however, gives rise to practical problems. Although less direct supervision by the instructor is required when students use a programming language, the amount of student-generated material he must deal with is very much greater. And the occasional help with "debugging" and extending programs that the instructor must provide requires that he easily and efficiently follow each student's current work.

In practice the instructor's involvement must be substantially reduced -- there are typically far too many students for an instructor and lab assistants to handle in this way. This situation can, we believe, be substantially improved. Our contention is that the great manipulative power the computer provides the students can also be made accessible to the instructor for more efficiently monitoring and analyzing the students' work. This report describes the design and development of a system directed to this end.

Part 1 of the report is a user's guide and overview of the system. Part 2 is a detailed description of the system including full program documentation; Part 3 contains standard analysis packages developed to facilitate applications; Part 4 contains examples of system use. A technical description of the LOGO language is appended.

Part 1.

User's Guide

1. Programming Languages in Teaching

Programming languages are coming into extensive use in undergraduate instruction. In addition to languages such as FORTRAN, JOSS, and APL, originally designed for scientific applications, some, like BASIC and LOGO, were expressly designed for student use. These languages can be employed in a number of different ways, encompassing a great variety of teaching modes as well as student uses. In some situations the computer is used primarily for evaluating complex expressions; sometimes it is used for parametric study of previously-defined models; sometimes for exploring complex data structures; sometimes the student develops his own programs in connection with specific course projects; and sometimes the programming language is used as the conceptual framework for developing some of the main ideas in the subject taught. These forms serve to illustrate the diversity of approach possible. A balanced design will combine several of them in a single course or even in the study of a single topic.

Utilization of a computer and a programming language does not, in itself, impose the need for new teaching strategies. Practical problems develop only in those cases where students are expected to carry out their own programming projects, whether or not there is extensive guidance. Experience shows that, here, great diversity and individuality in work arises and there is considerable divergence among students even when they are nominally working on the same problem. One consequence is that student errors can lie much deeper and be correspondingly more difficult to diagnose. Further, the teacher must understand the implications of a large number of alternative approaches to the solution of a given problem.

A good teacher always faces these difficulties. But in the teaching situations that develop from this way of using programming languages, his difficulties are made particularly critical by the amount of individual monitoring required. To examine this problem more concretely, it is useful to see the characteristics of actual student programming work in some detail; we will therefore study student work in part of a programming-oriented sequence used in undergraduate instruction (at the University of Massachusetts in 1971). To provide a context, we first outline the central ideas developed in the teaching guide associated with this undergraduate sequence.

2. An Example of Use of the LOGO Language

A programming language has a *central* use in the teaching discussed here. We illustrate this role next with a specific example. We designed the following sequence, and others like it, as models for guiding teachers. This particular sequence served that role in an introductory course in undergraduate mathematics given at the University of Massachusetts, Boston, in the spring 1971 term. The course was open only to students of low mathematical ability. Thus, we chose mathematical material which, at least on the surface, appeared nonmathematical.

In this sequence we trace the development of some programs for making pictures, specifically geometric figures, on a teletypewriter. We use the LOGO programming language* for this because we feel that crucial formal and heuristic issues are more clearly exposed with LOGO than with other programming languages. We begin by writing straightforward procedures for drawing figures of fixed shape and size. The first such procedures are pointillistic, each command typing out a single point of the figure. This is a tedious and mathematically uninteresting approach. A considerable improvement comes about from noting that we can write a procedure for typing out the figure a whole line at a time. This procedure is called MARK :N: :X: and is written entirely in terms of LOGO primitives. Here is its definition.

---

*A description of the LOGO language is given in Appendix A.

```
TO MARK :N: :X:
1 TEST IS :N: 0                (When there are no more :X:'s to
2 IFTRUE STOP                  type, stop.)
3 TYPE :X:                     (Otherwise, type out an :X:)
4 MARK (:N:-1)                 (And repeat the procedure :N:-1
END                            more times.)
```

For example:

```
MARK  18  "+"                  (We underline the student's typing)
++++++++++++++++++
```

We can write procedures using MARK to draw geometric figures of many different kinds. We can easily generate figures with vertical symmetry about some fixed line, by extending MARK to handle the details of formatting. The procedure MIDDLE :N: :X: neatly centers the row of :N: :X:'s in the middle of the line.

```
TO MIDDLE :N: :X:
1 MARK (QUOTIENT OF (60-:N:)   (Indent the appropriate number
   AND 2) :BLANK:              of spaces)
2 MARK :N: :X:                 (and then type out :N: :X:'s)
END
```

Thus:

```
MIDDLE  18  "+"
                    ++++++++++++++++++
```

MIDDLE has direct and straightforward application for writing procedures for drawing general classes of simple symmetric figures -- triangles, rectangles, and trapezoids of varied sizes and shapes. For example, we write a procedure TRAPEZOID and use it to make drawings such as the following.

TRAPEZOID 3 11 2

```
    +++
   +++++
  +++++++
 +++++++++
+++++++++++
```

TRAPEZOID 13 5 -2

```
+++++++++++++
 +++++++++++
  +++++++++
   +++++++
    +++++
```

We can use such procedures as the basis for further drawing procedures. Here, for example, is a procedure for drawing octagons, where :A: is the top base, :B: is the middle diameter, and :STEP: is the step size.

```
TO OCTAGON :A: :B: :STEP:
1 TRAPEZOID :A: :B: :STEP:
2 RECTANGLE :B: (QUOTIENT OF (:B:-:A:)
     AND :STEP:)
3 TRAPEZOID :B: :A: (-:STEP:)
END
```

A next level of procedures generates sequences of figures -- hexagons of increasing size, oscillating diamond patterns, and the like. For example, we write a procedure PATTERN which uses a random process to create a procedure for drawing patterns that are symmetric about a horizontal line as well as a vertical one. The following drawings were made by procedures generated by PATTERN.

```
            +
           +++
            +                                   +
        +++++++++                              +++
         +++++++                              +++++
          +++++                              +++++++
           +++                              +++++++++
          +++++                            +++++++++++
        +++++++++                        +++++++++++++++
      +++++++++++++                    +++++++++++++++++++
    +++++++++++++++++                +++++++++++++++++++++++
  +++++++++++++++++++++            +++++++++++++++++++++++++++
+++++++++++++++++++++++++        +++++++++++++++++++++++++++++++
  +++++++++++++++++++++            +++++++++++++++++++++++++++
    +++++++++++++++++                +++++++++++++++++++++++
      +++++++++++++                    +++++++++++++++++++
        +++++++++                        +++++++++++++++
          +++++                            +++++++++++
           +++                              +++++++++
          +++++                              +++++++
         +++++++                              +++++
        +++++++++                              +++
            +                                   +
           +++
            +
```

Using the procedures written thus far, the student could generate a large number of different geometric shapes, and the writing of such a sequence of procedures represents a considerable achievement on his part. The foundations of geometry, however, lie in the transformation of geometric objects, not merely in their portrayal. Our next goal, then, is to write procedures for performing standard transformations of geometric figures. Such transformations include translation, rotation, and reflection, both with respect to a given point and with respect to a given line. Procedures to generate the union and intersection of the sets of points defining two geometric objects are also useful. And, we need a procedure DRAW which plots any given set of points.

To write these procedures we need a different representation for geometric objects, one which can be retained within the computer. (Clearly we do not have such a representation thus far -- our current objects are generated and drawn one line at a time.) Perhaps the simplest such representation is a list of pairs of numbers, each pair representing one point of the object. Then it is easy to write procedures, such as the following, which reflect a set of points about the x-axis.

```
TO REFLECTX :PAIR LIST:           (:PAIR LIST: is the list of X Y
                                  number pairs)
1 TEST IS :PAIR LIST: :EMPTY:     (Are there any points left on
2 IFTRUE OUTPUT :EMPTY:           :PAIR LIST:?  If not, terminate
                                  procedure)
3 OUTPUT LIST OF
    FIRST OF :PAIR LIST:
    NEGATIVE OF SECOND OF :PAIR LIST:
    REFLECTX OF (BUTFIRST2 OF :PAIR LIST:)
                                  (Otherwise, output a list of the
                                  X coordinate and the negative of
                                  the Y coordinate of the first
                                  number pair on :PAIR LIST:, and
                                  REFLECTX applied to the pair list
                                  obtained by deleting the first
END                               number pair on :PAIR LIST:)
```

Thus, for example:

```
PRINT REFLECTX OF "1 2 4 3 7 11 2 -1"
1 -2 4 -3 7 -11 2 1
```

Using this and two similar procedures, one for reflecting about the 45 degree line through the origin, and the other for reflecting about the Y-axis, we can now write a procedure for random generation of figures having eightfold symmetry.

```
TO EIGHTFOLD :N:                (:N: is the number of pairs on the
                                pair list that will be generated)
1 MAKE "PAIR LIST"              (Generate a random list of :N:
   RANDOMLIST OF :N:            pairs, :PAIR LIST:)
2 MAKE "PAIR LIST"              (Form the union of :PAIR LIST: and
   UNION OF (:PAIR LIST:) AND   the pair list formed by reflecting
   (REFLECT45 OF :PAIR LIST:)   it around the 45 degree line, and
                                make this the new :PAIR LIST:)
3 MAKE "PAIR LIST"              (Form the union of the new list and
   UNION OF (:PAIR LIST:) AND   its reflection about the Y-axis)
   (REFLECTY OF :PAIR LIST:)
4 MAKE "PAIR LIST"              (Do the same with the resulting list
   UNION OF (:PAIR LIST:) AND   and its reflection about the X-axis)
   (REFLECTX OF :PAIR LIST:)
5 DRAW ORDER OF :PAIR LIST: "+" (Put the resulting pair list in
                                lexicographic order and plot it
                                using +'s)

END
```

EIGHTFOLD generates random drawings such as the following.

```
   + +                            +               +
   +++                           +  +           +  +
    +
++     ++                         +  +   + +   +  +
 ++   ++                            +           +
++     ++
    +
   +++                              +           +
   + +
                                    +           +


                                    +           +
                                  +  +   + +   +  +

                                 +  +           +  +
                                  +               +
```

Using such basic transformation procedures we can study general Euclidean transformations. For example, we can develop methods for determining whether two randomly oriented polygons are congruent. Proceeding in this fashion we steadily progress from pre-mathematics to material of genuine mathematical content.

3. Dribble Files

What we just sketched was an "ideal" teaching sequence. Let us look next at what students actually did. We will not look at the entire development of a sequence of drawing programs, only at the form and organization of some nearly final versions derived from student interactions. Even in this condensed form, very specific information about individual student work, of interest for both research and teaching, is obtained.

We begin by looking at examples of real student interactions in raw form, obtained from "dribble files" generated by the student and recorded by the computer while he works. We then consider more generally the work that students did to see what conceptual and pedagogic issues arise in the course of that work.

The computer can present the student's work to the instructor in various ways. One important mode of presentation is periodic listings of the student's own program files. The student generally updates his program files each working session and these files contain nearly all the programs he writes. The instructor needs to know, however, not only the programs generated by the student but also how these were written, debugged, edited, and used.

One way of giving him this information is through hard-copy transcripts of the student's entire interaction with the computer. These can be obtained simply by using two-copy paper in the teletypewriter. All the required information is acquired naturally and inexpensively in this way.

Unfortunately, the information thus obtained is not in a form that can easily be used. To make this information available for computer retrieval and processing, we need to store it as it is produced. Such stored files of the student's entire dialogue with the computer, generated as he works, are called "dribble files". These files contain all the student's typed inputs, not just the programs and data that he himself chooses to store.

Each typed input is given a "time stamp" -- i.e., we record with the input line the total elapsed time from the entry of the previous line to the entry of the current line. Thus, we can do latency analyses as well as other processing involving considerations of time. We have extended the LOGO language processor to enable creation of dribble files as a by-product of student work. In these dribble files, we store the student's type-ins but not the associated computer responses. (The responses can easily be regenerated later.)

In the next pages, we shall discuss the content and the uses of such dribble file information. Dribble files of student programming generated in introductory mathematics course work at the University of Massachusetts will serve to illustrate the discussion. The following example shows the listing of a fragment of a dribble file made from the work of RC, one of the students. The file is identified on the top line: RC.DRB;2 along with the date and time it was generated. We have prefixed the lines with reference numbers 0 through 17. Each line starts off with the time stamp. Thus, in line 0 the number 0:00:10 means 0 hours 00 minutes and 10 seconds of time required to complete this line.

The material concerns the development of a procedure for drawing triangles. Lines 0 and 1 direct the definitions of the procedures NUM and TRIANGLE to be listed. The resulting printouts are shown on the right. The TRIANGLE procedure is edited several times during this session: in lines 2, 3, and 4; later in lines 6, 7, 8, and 9; and once again in lines 12, 13, and 14. In between these successive editing modifications. the effect of the changes made in TRIANGLE is tested by executing the procedure NUM with the input 8. This is done in line 5, then in line 11, and finably in line 15. The computer printouts from the executions - the various drawings - are shown on the right. At the end of this exchange (line 16), the final version of TRIANGLE (which still has a "bug" in its stopping rule) is listed. In line 17 NUM and TRIANGLE are stored in their current forms in a file labeled "JOHN CAD".

```
DRIBBLE FILE LINE                 EFFECT AS EXECUTED

RC.DRB;2  THU 29-APR-71 12;40PM

0 0.00:10 LIST NUM                TO NUM :N:
                                  10 MAKE "NU" 1
                                  20 TRIANGLE :N:
                                  END

1 0:00:11 LIST TRIANGLE           TO TRIANGLE :N:
                                  10 MARK "X" :NU:
                                  20 PRINT :EMPTY:
                                  30 TEST IS :NU: :N:
                                  40 IFTRUE STOP
                                  50 MAKE "NU" SUM OF :NU: AND 1
                                  60 TRIANGLE :N:
                                  END

2 0:00:10 EDIT TRIANGLE
3 0:00:43 10 MIDDLE 50 "X" :NU:
4 0:00:05 END
5 0:00:09 NUM 8                         X
                                        XX
                                       XXX
                                       XXXX
                                      XXXXX
                                      XXXXXX
                                     XXXXXXX
                                     XXXXXXXX
```

```
6 0:01:05 EDIT TRIANGLE
7 0:00:26 50 MAKE "NU" SUM :NU: 2
8 0:00:25 30 TEST IS SUM :NU: :N:
9 0:00:03 END
10 0:00:09 NUM8\8                NUM8 NEEDS A MEANING
11 0:00:07 NUM 8                 X

                                 SOMETHING MISSING.  IS NEEDS
                                 ANOTHER INPUT.  I WAS AT LINE 30
                                 IN TRIANGLE
12 0:02:49 EDIT TRIANGLE
13 0:00:17 30 TEST IS SUM :NU: 2 :N:
14 0:00:02 END
15 0:00:06 NUM 8                       X
                                      XXX
                                     XXXXX
                                    XXXXXXX
                                   XXXXXXXXX
                                  XXXXXXXXXXX

                                 WHEN YOU STOPPED ME I WAS AT
                                 LINE 50 IN TRIANGLE

16 0:00:38 LIST TRIANGLE         TO TRIANGLE :N:
                                 10 MIDDLE 50 "X" :NU:
                                 20 PRINT :EMPTY:
                                 30 TEST IS SUM :NU: 2 :N:
                                 40 IFTRUE STOP
                                 50 MAKE "NU" SUM :NU: 2
                                 60 TRIANGLE :N:
                                 END

17 0:01:00 SAVE JOHN CAD
```

We could have illustrated the use of dribble files with more complex and technical examples such as from a sequence on formal differentiation in calculus. Instead, we have chosen to work with material derived from actual classroom teaching. As is evident from the above examples, the underlying issues of program form and organization arise even with relatively simple undergraduate material. Moreover, they do not essentially depend upon the programming language used.

If another programming language such as BASIC had been used instead of LOGO, the same organizational concepts would have appeared, though in somewhat different form. To illustrate, we include a triangle-drawing program written in BASIC. Though this program superficially appears quite different from the corresponding set of LOGO procedures, the two forms are very close. The process again falls into three parts -- the main part draws the triangle iteratively, one line at a time; this part calls a subroutine which, like MIDDLE, draws a centered line; and this subroutine itself twice calls another subroutine which, like MARK, types a row of characters. Within each part, of course, there are formal differences from the corresponding LOGO procedures. Iteration is performed here using FOR statements rather than by simple recursion, and all variable names have to be treated as global. The use of the program is shown following the program listing.

```
                                   (Corresponds to TRIANGLE
                                   using character A$ and having
                                   L rows)
10 INPUT A$
20 INPUT L
30 FOR K=1 TO 2*L+1 STEP 2
40 GOSUB 9000
50 NEXT K
60 GO TO 9999
                                   (Corresponds to MIDDLE,
                                   centers K markings of A$)
9000 LET C$=" "
9010 LET N=25-K/2                  Marks 25-K/2 blank spaces
9020 GOSUB 9100
9030 LET C$=A$
9040 LET N=K                       Marks A$, K times
9050 GOSUB 9100
9060 PRINT                         (Goes to the next line)
9070 RETURN
```

```
                                   (Corresponds to MARK, types
                                   C$, N times)
9100 FOR J=1 TO N
9110 PRINT C$;
9120 NEXT J
9130 RETURN

9999 END


READY
RUNNH

?*
?8

                       *
                      ***
                     *****
                    *******
                   *********
                  ***********
                 *************
                ***************
```

4. Building a Language and Monitor System for Processing Dribble Files

The "dribble file" we have described contains all details of the student-computer interaction as it occurs at the teletypewriter. By *replaying* a dribble file, we can even get all the information at the systems level. Thus, the dribble file certainly contains all the raw data available for analysis. The very completeness and bulk of the information in the dribble file, however, discourage us from doing any searching and processing directly. We could have collected the data selectively to reduce the size of the file but preselection of the data to be preserved can turn out badly. Furthermore, any preselection rules can be applied to the dribble file itself which can then be saved as a backup. With this strategy, if it turns out in light of consequent results that a poor choice has been made, a new "preselection" can be done on the dribble file. This is our rationale for saving all the data.

Since it is inefficient to use dribble files directly, we must ask what aids exist or can be devised to make their use manageable. The most rudimentary such aid is a text-editing language, such as TECO, as implemented on the PDP-10 computer system. Direct character-by-character matching is made very easy by such a language. Thus, for example, one could delete all time marks in a given file or all directly executed input lines. One could also delete all lines followed by an error message, if one can specify the format of an error diagnostic statement. These actions are all the results of simple format matching. Also, it is easy for a user to insert comments into a dribble file using TECO. If, in addition, one can combine series of the basic searching, inserting, deleting, and pointer-moving commands, with numeric and branching capabilities, there is the possibility of extremely sophisticated types of processing. In fact, the "Q-registers" that TECO provides for storing stacks make the language perfectly general and permit Turing-machine-like programs to be written for all computable functions. One could, for example, with some effort write a program using TECO to find and enumerate all simple recursive programs in a dribble file.

Unfortunately, in a practical sense this is about as far as one can go with TECO. First, one is writing programs in what is essentially machine-language, a rather tedious undertaking. Also, it is difficult to write programs that are easily extensible.

Thus, two requirements that a dribble file analysis language must satisfy are already apparent. The language itself must be natural in form to accommodate the unsophisticated user, and user-written procedures must be transparent to permit their use in further procedures. It is clear that such a language should sppropriately incorporate the text-handling and editing features which are already in common use. By making the language self-extensible, so that sets of programs of any depth can easily be written, we satisfy the requirement of transparency. Also, it is much easier to write general programs in a self-extensible language. For example, instead of writing a TECO program which looks in the dribble file for an object of some given form, one can, with about the same effort, write a program in such a language, one of whose inputs specifies the form to be found. Also, a SNOBOL-like set of matching procedures could be written in the file analysis language itself instead of being given as part of the set of primitives.

Let us discuss the use of the analysis language next. Often a teacher wishes to classify the programs written by his student in a way he specifies. It would clearly be very inefficient to have to perform this analysis on the same programs each time he looked at the dribble files. The standard result of an analysis of a dribble file should therefore be a new file containing the processed data, with tags joining it to the original file at points of correspondence. This means that the user can look through the processed file using his own set of descriptors and can go back to the raw data whenever necessary.

The idea of being able to operate upon the file at multiple levels of detail is of very general use. In analyzing the work of a student through his dribble files, there are several levels which may be of nearly simultaneous interest. For the top level, a good mode of presentation might be a flow chart, dynamically changing as the user scans the student's work, indicating all the programs in the student's workspace and showing the changing connections between them. At a lower level one might have a complete specification of all the student's programs at that moment in time. At the lowest level, one would probably want a "cleaned up" version of the dribble file with (what the user considers) the obscuring features deleted. The analysis of the dribble file would begin at the top level. When programs of particular interest appeared, they might be listed or executed. Still further, the details of their creation and use by the student might be explored at the lowest level.

Thus, we see the need for a set of programs enabling the user to switch back and forth between levels, zooming in when he needs more information, allowing him to vary his scanning rates, to go back and forth between current and previous material, to switch from scanning to execution mode, and so on. Let us consider next some kinds of information that will be of interest.

Apart from considerations specific to the content being studied, the user probably will be interested in general questions regarding the formal structure and organization of the student's work. Examples are: (1) What kinds of programs were used, i.e., what standard functions did the programs have (such as initialization, testing, and computation)? (2) What elementary program forms were used (loop-free sequence, iteration, simple recursion, etc.)? (3) What was the program organization, i.e., how were the various programs combined (program tree, substructure type, recursion diagram)? Thus, we can characterize the functional, formal, and organizational features of the work or particular students.

For example, it is useful to follow the evolution and mastery of a given program form in the work of an individual student. One student, RC, early in the term was confronted by the need for a procedure to find the integral half of a number. Her algorithm consisted of successively adding 1 to a trial "half" and testing to see whether its double was within 1 of the original number. Having, after considerable effort, written the recursive procedure FIND to do this, she then saw the need for another program to do the initialization and wrote HALF. Annotated listing of both programs are given following. They fall neatly into distinct parts as labeled. The algorithm itself is, perhaps, not one that a more sophisticated programmer would use. Also, in many places RC is more obscure than is necessary. Real student programs are like this, however.

```
                 TO HALF :N:                      (:N: is the number
                                                  to be halved)
Initialize       10 MAKE "TRIAL" 0                (Set the "trial"
                                                  value of half to 0)
Call Simply-     20 OUTPUT FIND OF :N:            (Output the result
Recursive                                         of FIND as the
Procedure                                         answer)
                 END


                 TO FIND :N:

End-Test         10 TEST GREATERP OF 2            (Is 2*:TRIAL:
                    AND DIFFERENCE OF             within 1 of :N:)
                    (:N:) (PRODUCT 2 :TRIAL:)
                 20 IFTRUE OUTPUT :TRIAL:         (If so, :TRIAL: is
                                                  the answer
Increment        30 MAKE "TRIAL"                  (Otherwise, add 1
                    (SUM OF :TRIAL: AND 1)        to :TRIAL:)
Recursion        40 OUTPUT FIND OF :N:            (and repeat FIND)
                 END
```

About a week later the same student wrote a pair of programs to automatically draw triangles. We showed the dribble file for the last part of this development in Section 3. The form of these programs is nearly identical to the ones for halving. The only change is that each step of the recursive procedure TRIANGLE results in an action and this was not true for FIND. This task, however, only took about half the time required for the earlier one. Along with this, the problem was approached much more directly, as is evident from looking at the dribble file. Clearly, this program form was being internalized.

```
               TO NUM :N:                     (:N: is the number
                                              of X's in the bottom
                                              row of the triangle)
Initialize     10 MAKE "NU" 1                 (Make the number of
                                              X's in the current
                                              row 1)
Call Simply-   20 TRIANGLE :N:                (Draw the triangle)
Recursive
Procedure
               END


               TO TRIANGLE :N:

Action         10 MIDDLE 50 "X" :NU:          (Mark the current
                                              row)
               20 PRINT :EMPTY:               (Start the next row)
End-Test       30 TEST IS (SUM OF             (Is this the last
                  :NU: 2) (:N:)               row?)
               40 IFTRUE STOP                 (If so, done)
Increment      50 MAKE "NU"                   (Otherwise, get
                  (SUM OF :NU: 2)             number of marks in
                                              current row)
Recursion      60 TRIANGLE :N:                (and repeat
                                              TRIANGLE)
               END
```

This example forms a small part of RC's work on the geometric figure drawing sequence. In all, she used three different program forms: the one which we have just discussed which we will call form II; simple recursion which we label form I; and form 0 which is a linear sequence of steps. The diagram in Figure 1 shows all the connections between the various parts of the drawing sequence. The program forms are indicated in parentheses after each procedure name. (A more complete "flow chart" would show the conditions for recursion and termination of each procedure of form I or II. This information has been omitted, however, for the sake of clarity and conciseness.)

Another student was working on programs for drawing geometric figures during the same period. The diagram associated with the work of this student, AF, is shown in Figure 2. Great differences in program organization in the two cases are apparent, even though the set of programs have the same final effect.

These examples show some of the issues involved in analyzing complex student interactions. These students spent about three weeks near the beginning of the term writing these programs. Thus it is apparent that complex structures can be generated quickly, even by "beginners". We need to consider next how such structures are retrieved from dribble files.

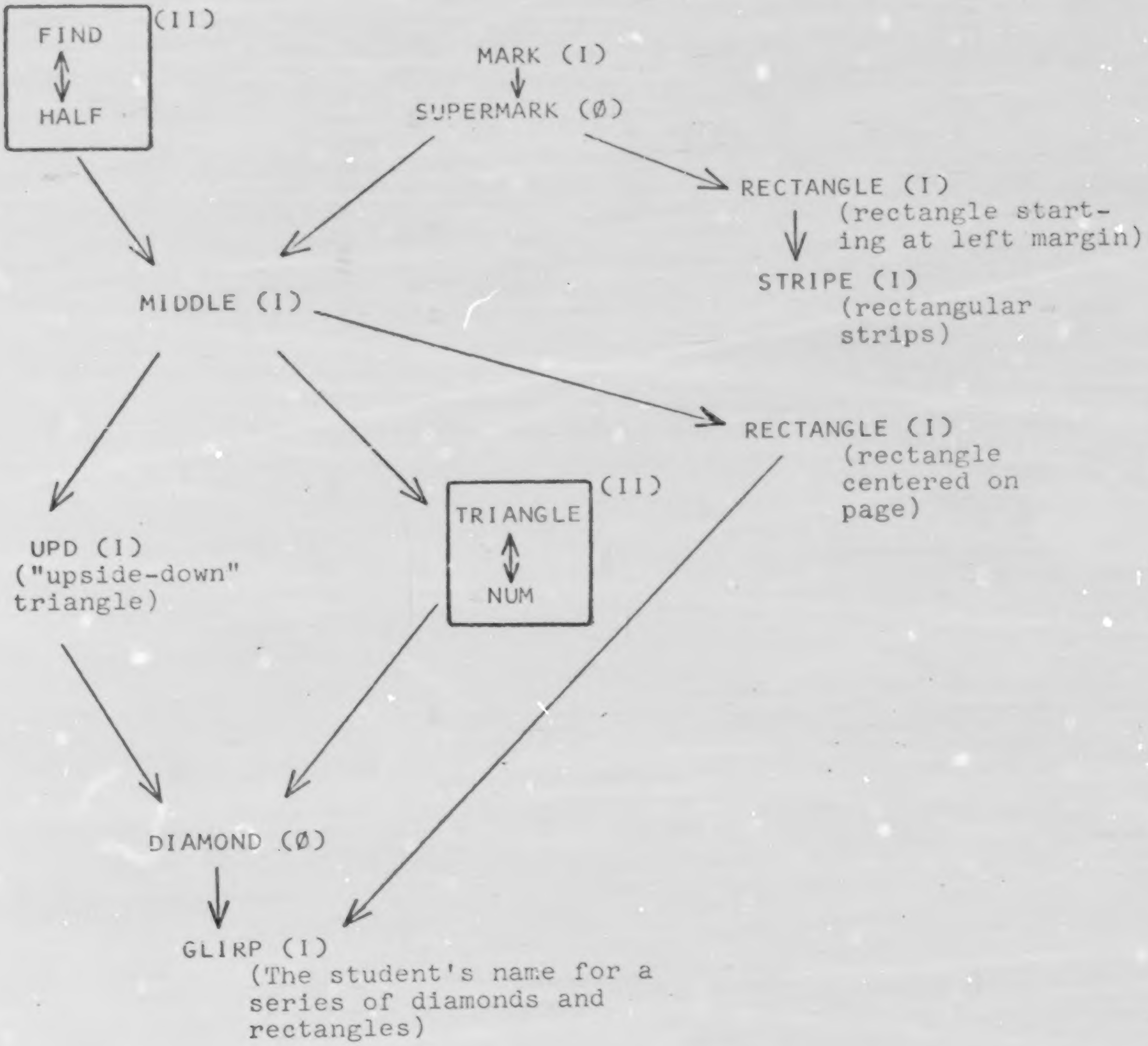

FIND
HALF
(II)
MARK (I)
SUPERMARK (0)
RECTANGLE (I)
(rectangle starting at left margin)
STRIPE (I)
(rectangular strips)
MIDDLE (I)
RECTANGLE (I)
(rectangle centered on page)
UPD (I)
("upside-down" triangle)
TRIANGLE
NUM
(II)
DIAMOND (0)
GLIRP (I)
(The student's name for a series of diamonds and rectangles)

Figure 1. Diagram of RC's Drawing Program

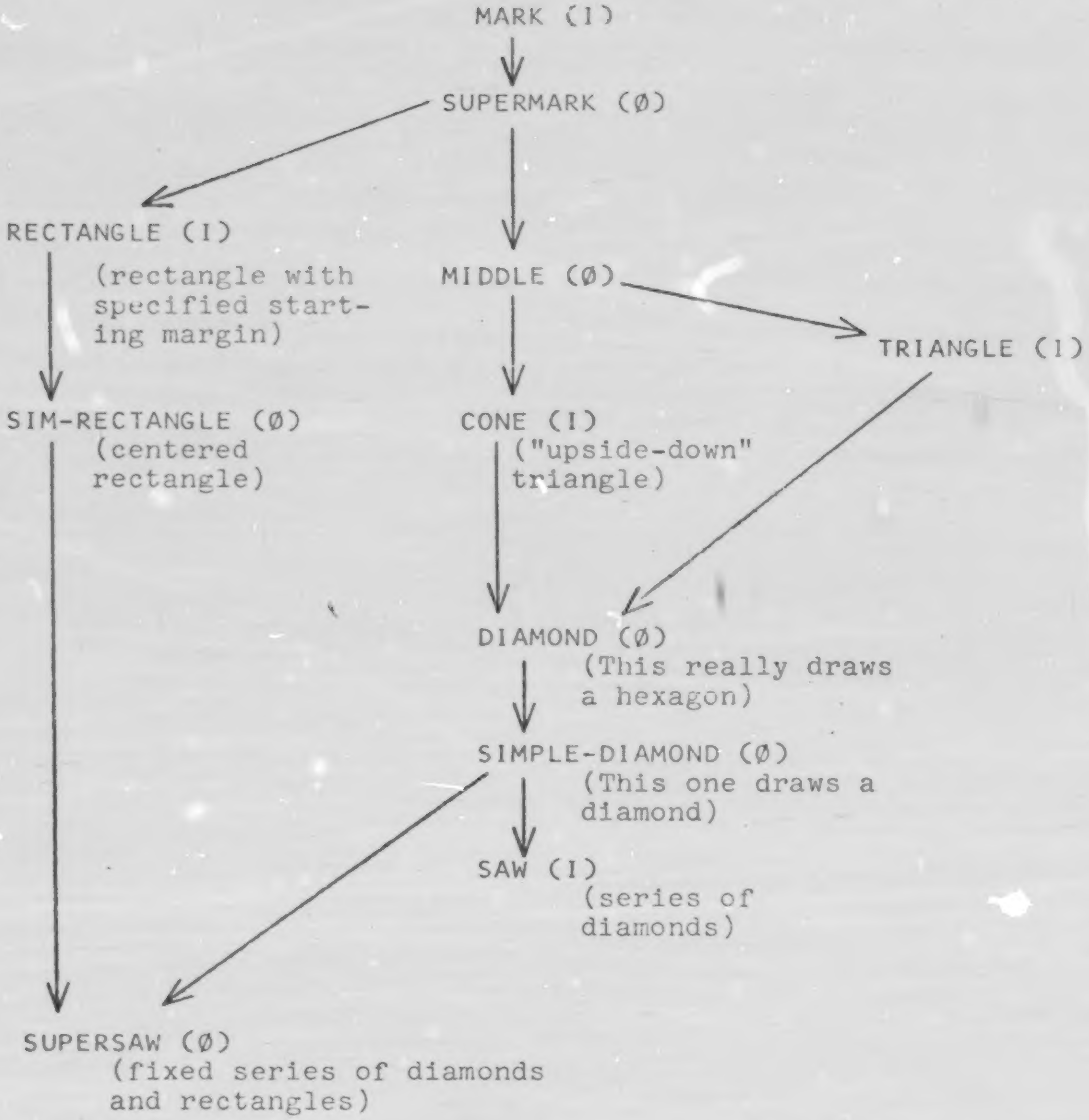

MARK (I)
SUPERMARK (0)
RECTANGLE (I)
(rectangle with specified starting margin)
MIDDLE (0)
TRIANGLE (I)
SIM-RECTANGLE (0)
(centered rectangle)
CONE (I)
("upside-down" triangle)
DIAMOND (0)
(This really draws a hexagon)
SIMPLE-DIAMOND (0)
(This one draws a diamond)
SAW (I)
(series of diamonds)
SUPERSAW (0)
(fixed series of diamonds and rectangles)

Figure 2. Diagram of AF's Drawing Programs

5. The Raw Dribble File

In this section we begin a detailed discussion of the processing of dribble files by first describing their structure. A dribble file is simply the very slightly altered input stream produced in the course of a single LOGO session. This input stream is generated both by teletype inputs as well as retrievals of previously SAVEd LOGO files. When a version of LOGO containing the dribble file generator is entered, the user is asked for identification -- INITIALS PLEASE. Entering the initials NMI aborts the dribble file generation process. Any other input is acceptable and is used as the name of the dribble file which is subsequently generated. (In our initial use of dribble files in teaching in spring, 1971, each student followed his initials with a digit to enable more than one file to be created under his initials on any given day. To avoid the overwriting and clobbering of previous files of the same student, automatic numbering was later implemented.)

A sample segment of a dribble file is given in Table 1. This file was created during a student working session. Associated with the file is a header which gives the date and time the file was created as well as the identifier entered by the student. The input stream following has been augmented by time marks at the beginning of each line; these give the elapsed time since initiation of the previous line. These time marks and the heading are the only modifications provided to the raw input.

Two aspects of dribble files can easily be noted by inspection of Table 1. First, the special time mark 24:00:00 precedes lines which were retrieved from storage, as opposed to

```
24:00:00
24:00:00 TO FIND :N:
24:00:00 30 TEST GREATERP "2" DIFFERENCE DIFFERENCE :N: :TRIAL: :TRIAL:
24:00:00 40 IFTRUE OUTPUT :TRIAL:
24:00:00 50 MAKE "TRIAL" SUM :TRIAL: "1"
24:00:00 60 OUTPUT FIND :N:
24:00:00 END
24:00:00
24:00:00 TO HALF :N:
24:00:00 10 MAKE "TRIAL" "0"
24:00:00 20 OUTPUT FIND :N:
24:00:00 END
24:00:00
24:00:00 TO DELETE :CHAR: :N:
24:00:00 10 TEST IS DIFFERENCE :N: "1" "0"
24:00:00 20 IFTRUE OUTPUT BUTFIRST :CHAR:
24:00:00 30 OUTPUT DELETE BUTFIRST :CHAR: DIFFERENCE :N: "1"
24:00:00 END
24:00:00
24:00:00 TO STRIPE :M: :N: :Y: :S:
24:00:00 1 SUBRECTANGLE :M: :N: :Y:
24:00:00 20 SUBRECTANGLE :M: :N: SUM :Y: "1"
24:00:00 30 TEST IS SUM :Y: "1" :S:
24:00:00 40 IFTRUE STOP
24:00:00 50 STRIPE :M: :N: SUM :Y: "1" :S:
24:00:00 END
24:00:00
24:00:00 TO MIDDLE :N: :X:
24:00:00 10 MARK "#" DIFFERENCE HALF :N: HALF COUNT :X:
24:00:00 20 TYPE :X:
24:00:00 END
```

All of the lines with 24:00:00 time marks were retrieved sequentially from a LOGO file previously saved by the student. Their effect is to load the LOGO procedures FIND, HALF, DELETE, STRIPE, and MIDDLE into the workspace.

```
24:00:00
24:00:00 MAKE↑T ↑T
24:00:00
0:00:10 TO TRIANGLE :N:
0:00:17 TRIANGLE 3
0:01:13 EDIT TRIANGLE
0:00:18 10 MARK "X" 1
0:00:33 40 MARK "X" SUM "X" 1
0:00:07 END
0:00:15 TRIANGLE 3
0:00:25 LIST TRIANGLE
0:01:38 EDIT TRIANGLE
0:00:20 20 T IS SUM 1 1 :N:
0:00:25 40 MARK "X" SUM 1 1
0:00:10 END
0:00:11 TRIANGLE 3
0:00:57 LIST TRIANGLE
0:03:20 LIST SUBRET0:00:16 LIST SUBRECTANGLE
0:00:25 LIST SUPERMARK
```

The remaining lines are student inputs to LOGO. The student is clearly involved in running, debugging, and editing a LOGO procedure, TRIANGLE.

TABLE 1. A "Raw" Dribble File Segment

those lines directly typed in. (Distinguishing between lines which were typed in from those which were retrieved is useful to the dribble file analyzer. More about this later on.)

The other thing that should be noted is the appearance of occasional time marks in the middle of lines. This occurs when an input line has been terminated with a RUBOUT, rather than an EOL character, meaning that the user has aborted the input of that line. The next line of input is then continued on the same line since no EOL* appears in the dribble file.

A student session usually commences with a GET command, with which the student retrieves his work at the point he left off. The partial example, given as Table 1, represents about one fourth the length of a typical session involving beginning users. As the student progresses, his dribble files reach lengths of 10 pages or more for a 90 minute session. Our choice of recording only the input stream (instead of both input and output) arose largely from the great amounts of data generated in such sessions. We retain the minimum amount of information required to completely reconstruct the student work at any point. Inclusion of the output stream would make the dribble file more readable, but would not be at all useful in such reconstruction. Such inclusion would much more than double the amount of information recorded, would lead to minor technical problems in mixing streams, and would require more CPU time in its generation.

---

*The end-of-line character.

6. Preprocessing of the Raw Dribble File

The raw dribble file contains a good deal of information which is of marginal interest for most analysis and which therefore can be eliminated. Character-by-character editing in the student's work, for example, is of no general interest other than in statistical studies--the student has instantly corrected his typing errors without requiring feedback. Also, the raw dribble file has to be massaged to put it into a pattern transformable to LOGO data types. Preprocessing encompasses these two activities. It is implemented in the TECO text editing language, the actual programs being given in Part 2.

Cleaning up of the files involves chiefly elimination of explicit line editing and spurious characters such as illegal control characters. Also, the various nonspace separators are spaced out for easy decomposition later on. Empty dribble file lines are removed. Multiple spaces are eliminated automatically later on, so these are ignored.

In order to be useable within LOGO, the successive lines of the dribble file are assigned as values of variables created by the preprocessing. At this level we must generate the actual assignment statements for subsequent execution. The statements have the form

```
MAKE ↑T var name ↑T ↑T var value ↑T,
```

where ↑T delimits both the variable name and the variable value. (These control T's are converted to quotes in subsequent processing.) When these assignment statements are read into a LOGO workspace with a GET command, they are executed and the bindings actually made. The variable names are created in the form "(n) N" where (n) runs sequentially from 1 by an increment of 1, providing an ordering for the dribble file lines, and N is a literal denoting name. The value of each such variable is the actual content of the associated dribble file line. If the line is one retrieved from a pre-existing LOGO file (with time mark 24:00:00), an additional assignment statement is generated of the form,

```
MAKE ↑T (n) A ↑T ↑T STORED ↑T.
```

Use of this set of auxiliary variables will be discussed in the next section.

To use the preprocessing program, the user enters TECO and types `;Y LOADER.TEC⤸ HXA⤸ MA⤸`. This sequence of steps loads the loading macro and initiates its execution. The program asks the user for the name of the input file -- the one to be preprocessed and the name of the output file, in which the result is to be stored. Since, for our purposes, no useful information is destroyed during preprocessing, as a general practice we overwrite the output onto the input file. In Table 2 we give the preprocessed form of a roughly corresponding segment of the raw dribble material shown in Table 1. The first line of Table 2 is the beginning of the stored procedure DELETE. The last lines of the two tables correspond.

```
MAKE ↑T43 A↑T ↑TSTORED↑T
MAKE ↑T43 N↑T ↑TTO DELETE : CHAR : : N :↑T
MAKE ↑T44 A↑T ↑TSTORED↑T
MAKE ↑T44 N↑T ↑T10 TEST IS DIFFERENCE : N : " 1 " " 0 "↑T
MAKE ↑T45 A↑T ↑TSTORED↑T
MAKE ↑T45 N↑T ↑T20 IFTRUE OUTPUT BUTFIRST : CHAR :↑T
MAKE ↑T46 A↑T ↑TSTORED↑T
MAKE ↑T46 N↑T ↑T30 OUTPUT DELETE BUTFIRST : CHAR : DIFFERENCE : N : "
                                          ↑↑1 "↑T
MAKE ↑T47 A↑T ↑TSTORED↑T
MAKE ↑T47 N↑T ↑TEND↑T
MAKE ↑T48 A↑T ↑TSTORED↑T
MAKE ↑T48 N↑T ↑TTO STRIPE : M : : N : : Y : : S :↑T
MAKE ↑T49 A↑T ↑TSTORED↑T
MAKE ↑T49 N↑T ↑T1 SUBRECTANGLE : M : : N : : Y :↑T
MAKE ↑T50 A↑T ↑TSTORED↑T
MAKE ↑T50 N↑T ↑T20 SUBRECTANGLE : M : : N : SUM : Y : " 1 "↑T
MAKE ↑T51 A↑T ↑TSTORED↑T
MAKE ↑T51 N↑T ↑T30 TEST IS SUM : Y : " 1 " : S :↑T
MAKE ↑T52 A↑T ↑TSTORED↑T
MAKE ↑T52 N↑T ↑T40 IFTRUE STOP↑T
MAKE ↑T53 A↑T ↑TSTORED↑T
MAKE ↑T53 N↑T ↑T50 STRIPE : M : : N : SUM : Y : " 1 " : S :↑T
MAKE ↑T54 A↑T ↑TSTORED↑T
MAKE ↑T54 N↑T ↑TEND↑T
MAKE ↑T55 A↑T ↑TSTORED↑T
MAKE ↑T55 N↑T ↑TTO MIDDLE : N : : X :↑T
MAKE ↑T56 A↑T ↑TSTORED↑T
MAKE ↑T56 N↑T ↑T10 MARK " " DIFFERENCE HALF : N : HALF COUNT : X :↑T
MAKE ↑T57 A↑T ↑TSTORED↑T
MAKE ↑T57 N↑T ↑T20 TYPE : X :↑T
MAKE ↑T58 A↑T ↑TSTORED↑T
MAKE ↑T58 N↑T ↑TEND↑T
MAKE ↑T59 A↑T ↑TSTORED↑T
MAKE ↑T59 N↑T ↑TMAKE↑T
MAKE ↑T60 N↑T ↑TTO TRIANGLE : N :↑T
MAKE ↑T61 N↑T ↑TTRIANGLE 3↑T
MAKE ↑T62 N↑T ↑TEDIT TRIANGLE↑T
MAKE ↑T63 N↑T ↑T10 MARK " X " 1↑T
MAKE ↑T64 N↑T ↑T40 MARK " X " SUM " X " 1↑T
MAKE ↑T65 N↑T ↑TEND↑T
MAKE ↑T66 N↑T ↑TTRIANGLE 3↑T
MAKE ↑T67 N↑T ↑TLIST TRIANGLE↑T
MAKE ↑T68 N↑T ↑TEDIT TRIANGLE↑T
MAKE ↑T69 N↑T ↑T20 T IS SUM 1 1 : N :↑T
MAKE ↑T70 N↑T ↑T40 MARK " X " SUM 1 1↑T
MAKE ↑T71 N↑T ↑TEND↑T
MAKE ↑T72 N↑T ↑TTRIANGLE 3↑T
MAKE ↑T73 N↑T ↑TLIST TRIANGLE↑T
MAKE ↑T74 N↑T ↑TLIST SUBRECTANGLE↑T
MAKE ↑T75 N↑T ↑TLIST SUPERMARK↑T
```

TABLE 2. A Segment of a "Preprocessed" Dribble File

7. Parsing of the Dribble File

In the previous section we have discussed preprocessing of "raw" dribble files. To initiate the processing phase, the preprocessed file must now be copied into LOGO-compatible form by use of the LOGO command COPY, which has the general form:

```
                                (FROM)
COPY (quoted system file name)  ( TO )  (unquoted LOGO filename,
     entryname)
```

For example, if the preprocessed file was given the name GLICK.LESSON1

```
COPY "GLICK.LESSON1" TO GLICK LESSON1
```

creates a LOGO file with roughly the same name as the preprocessed text file. If we now use the LOGO instruction LIST ENTRY GLICK LESSON1, we get almost exactly the same printout as from the (TENEX) system instruction LIST GLICK.LESSON1; the only difference being that the control-T's have become quotes. The important difference is that "invisible" heading information has been added so that the LOGO instruction GET GLICK LESSON1 results in the MAKE statements being executed and the appropriate bindings entered in the workspace ready for manipulation.

Following a COPY and a GET of the form described in the preceding section, the analyst has all the lines of the dribble file in his LOGO workspace. Each line could now be executed using the LOGO DO command, and a simple iterative procedure suffices to execute the dribble file lines to any desired point. Such a direct attack has both unfortunate as well as inconvenient consequences. First, one is not executing the statements of the dribble file in precisely the same environment in which they were first produced. In particular, a GET statement might not GET the entry fetched by the original execution of that command

and a SAVE might be disastrous, possibly clobbering a subsequently produced entry, replacing it with an outdated version. It is precisely for this reason that the input stream resulting from a GET has been incorporated into the dribble file. Both the GET and SAVE commands, therefore, should be no-ops, i.e., have no effect. Similarly, execution of GOODBYE might well be inconvenient. Also there is little point in executing lines with simple parsing errors.

A second difficulty in direct use of a dribble file lies in the generation of the information required for the graphical representation of procedure structure. Each procedure line has to be broken into elements and each element looked at to determine procedure interconnections. And, this process has to be repeated each time a structure is elicited further along in the file. Also, in a simple direct mode of dribble file access, many investigations such as following the evolution of a procedure definition would unnecessarily require execution of the entire dribble file segment.

These difficulties are easily resolved by the generation of auxiliary information for use in conjunction with the dribble file itself. Such information includes various data which expedite the execution of the dribble file such as data pointing out the non-utility of dribble file lines and, where appropriate, data to speed up graphical displays of procedure interconnections. The other useful type of information is concerned with answering analytic questions which really shouldn't require execution of the file. For example, a catalog of correctly parsed procedure lines enables the system to give the definition of any procedure at any point in the dribble file by a simple lookup procedure. (This cannot be done by simply looking for the definition lines directly because the "state" of the workspace must also be known.) Furthermore, as the user, through experience, evolves definite patterns of analysis of student dribble files, he can incorporate more and more of this analysis into procedures which generate desired information before the dribble file is scanned.

We call this phase the parsing phase, although parsing is not the only activity performed at this time. The purpose of the parsing phase is to make the execution and scanning of the dribble file efficient and simple at a reasonable cost in time. The general idea is to provide the analyst, and the analysis system, quick access to information which is likely to be required repeatedly in going through a dribble file while carrying out investigations.

The procedures making up our parser are given in Part 2. We give here only a summary of the parser's effects and structures. Each line is parsed and inspected element-by-element with the following effects: Some lines are not to be executed automatically, notably those with errors that might be fatal. For the line numbered "(number) N" an associated variable of the form "(number) D" is given data which label the line as having deleterious effects during execution (the user will usually wish to inhibit their execution during his analysis). Examples of such data are indicators for the commands GET, SAVE, and GOODBYE, as we just noted. Also included in this class are lines with simple parsing errors--such errors are not executed by LOGO and have no effect on the workspace. Also, in the absence of error-trapping capabilities, parsing of such lines always pops the user back to the top level of LOGO, inhibiting execution of whatever analysis program is being used.

Such annoyances can be avoided by skipping over badly formed lines. These errors, although chiefly parsing errors, can have a semantic flavor, however, because LOGO is in one of two states at any point - defining or not defining. The state is completely determined by finding which procedures have already been defined. Thus, we can get values for "(number) D" like "NOT DEFINING", and "FOO ALREADY DEFINED" in addition to local parsing error indicators such as "MISSING QUOTE". The general strategy is to suppress execution of all such lines, i.e., to skip over line "(number) N" if "(number) D" is not empty, although, as we see later, the decision to skip or execute can be conditional on the content of "(number) D".

As the parser goes through procedure definitions, a list of defined procedures is built up under the name "FINAL CONTENTS". The value of this variable is considered a list of triples, the first element of each being the procedure name, the second the dribble file line on which the procedure has been defined, and the third the number of inputs the procedure must take. This is the top level of the catalog mentioned above. At the next level down, for each procedure FOO, there is a variable "STRUC FOO" taken to contain a set of pairs. The first element of the pair is a dribble file line number, the second is the line of FOO which has been defined on that dribble file line. Thus, the evolution of any known procedure can be traced, using just "STRUC (pname)" and simple lookups, and a list of the procedures defined at any point can be easily determined using "FINAL CONTENTS". (At any point in time, we have, of course, not a true temporal referent, but the last line of the dribble file which has been, or is considered to have been, executed.)

The parsing package, as described, gives all information necessary to run through the dribble file and this routine operation is described in the next section. The user is also able to specify additional work to be done during parsing. In fact, as he gains experience with the system and his inquiries become more and more systematic, incorporation of his individual facilities might substantially modify the original parsing package and lead to the development of a customized semiautomatic system. To make such additions to the parsing procedures easily available to the user, we have included four special entry points in the form of the empty procedures $EXAMINELINE, $USEUPLINE, $EXAMINEEL, $USEUPEL.

$EXAMINELINE and $USEUPLINE are invoked as each succeeding line is first looked at. The difference between them is that $EXAMINELINE is to be a command which has no effect on subsequent parsing of the line; whereas $USEUPLINE must be a predicate, returning TRUE or FALSE in addition to any action it may perform. If $USEUPLINE returns TRUE, parsing of the line is inhibited and the next line is accessed.

Similarly, $EXAMINEEL is a command which is invoked as each element of the line is examined. $USEUPEL (like $USEUPLINE) is a predicate and it returns TRUE or FALSE corresponding to skipping to the next element or continuing. A fifth procedure $ENDLINE is invoked after the line is parsed to permit "cleaning up" of the work the user-defined procedures may have done.

Each of these empty procedures is "filled in" in the usual way, by defining a LOGO procedure with that name. If these empty procedures are not "filled in", parsing is carried out in the standard manner. These procedures, if wished, can be used to modify as well as examine the line being worked on--the name "CURRENT LINE" is given to the current line and its remaining portions as it is processed. Its number is :LINE NO:. Any of the five procedures may be defined to modify any of the global variables described earlier or, for that matter, to define new ones. In Part 3 we give two examples of definition of such procedures. One example deals with parenthesis-checking, the other with checking for command or operation.

An annotated listing of the parsing system is given in Part 2. A brief description of these procedures follows here, however, as an aid to the user who wishes to extend them. The top-level procedure is $PARSE which performs the necessary initialization, elicits a dribble file name, fetches it, and calls $GOTHROUGH to step through the lines of the dribble file. $STARTLINE then begins the actual parsing by passing the line to one of several procedures, depending on the type of line. Direct lines are immediately scanned, element-by-element, by $DOLINE; procedure definition lines pass first through $STOREDP. The other possible branches are the self-explanatory $PARSETO (for parsing procedure title lines), $PARSEEDIT, $PARSEEND, $PARSEERASE, $PLERASE (for line erasure, ERL), $PLEDIT (for line editing, EDL). These, in turn, call procedures which handle various levels of detail. $EXAMINELINE, $USEUPLINE, and $ENDLINE are, as might be surmised, contained in $STARTLINE and the procedures $EXAMINEEL and $USEUPEL in $DOLINE. Utility procedures include $GOODPARSEP :N: which checks emptiness of :(N) D:, i.e., whether the line passed muster or might need to be inhibited during execution, $ADDERROR which adds to :(N) D:, and $ADD :PLACE: :MES: which adds :MES: to the contents of :PLACE:. Table 3 is a summary of the variables used in parsing.

In generating the structure elements :(N) C: used for graphing student procedure structures, it is necessary to check

8. An Example of the Output of Parsing

In the following pages we show the results of $PARSE on a dribble file. The input is preserved--it is simply the "(n) N" and "(n) A" values. The structure variables cover the entire dribble file, though we show only part of the file in the example--we leave out some of the middle, as indicated. (It has been left unchanged by parsing.) Again the control T's (↑T's) are interpreted by LOGO as quotes. Table 3 summarizes the meanings of all variables used in the example shown in Table 4. Following the first three assignment lines, the middle portion repeats the output of the preprocessor (lines 1 N through 90 N; we show only the beginning and ending lines). The last part of the listing shows the final form of the parsed lines. We show all lines through 75 C, which corresponds to the last line of Tables 1 and 2.

```
MAKE ↑TCURRENT LINE↑T ↑T↑T
MAKE ↑TCURRENT PROC↑T ↑TTRIANGLE↑T
MAKE ↑TFINAL CONTENTS↑T ↑TTRIANGLE 1 1 MARK 9 2 SUPERMARK 15 4 RECTANG
                                   **LE 19 5 SUBRECTANGLE 26 3 FIND 33
                                   ** 1 HALF 39 1 DELETE 43 2 STRIPE 4
                                   **8 4 MIDDLE 55 2 NUM 87 1 TRIANGLE
                                   ** 90 1 TRIANGLE 176 1↑T
MAKE ↑T1 N↑T ↑TTO TRIANGLE : N :↑T
MAKE ↑T2 N↑T ↑T10 MARK " X "↑T
MAKE ↑T3 N↑T ↑T20 T IS SUM " X " 1 : N :↑T
MAKE ↑T4 N↑T ↑T30 IFT STOP↑T
MAKE ↑T5 N↑T ↑T40 MARK SUM " X " I↑T
MAKE ↑T6 N↑T ↑TEND↑T
MAKE ↑T7 N↑T ↑TTRIANGLE 3↑T
MAKE ↑T8 N↑T ↑TGET JOHN CAD↑T
MAKE ↑T9 A↑T ↑TSTORED↑T
MAKE ↑T9 N↑T ↑TTO MARK : CHAR : : N :↑T
MAKE ↑T10 A↑T ↑TSTORED↑T

          . . .
          . . .
          . . .
```

```
MAKE ↑T80 N↑T ↑T15 TYPE↑T
MAKE ↑T81 N↑T ↑TEDIT TRIANGLE↑T
MAKE ↑T82 N↑T ↑T15 TYPE " "↑T
MAKE ↑T83 N↑T ↑T15 PRINT " "↑T
MAKE ↑T84 N↑T ↑TEND↑T
MAKE ↑T85 N↑T ↑TERASE TRIANGLE↑T
MAKE ↑T86 N↑T ↑TTO : N :↑T
MAKE ↑T87 N↑T ↑TTO NUM : N :↑T
MAKE ↑T88 N↑T ↑T10 MAKE : N : 1↑T
MAKE ↑T89 N↑T ↑TEND↑T
MAKE ↑T90 N↑T ↑TTO TRIANGLE : M :↑T


MAKE ↑TSTRUC TRIANGLE↑T ↑T1 0 2 3 4 5 63 64 69 70 80 82 83 85 ERASE 90
                                   ** 0 91 92 93 94 95 111 123 125 126
                                   ** 127 143 144 160 ERASE 176 0 179
                                   **184↑T
MAKE ↑T2 C↑T ↑TTRIANGLE MARK↑T
MAKE ↑T3 C↑T ↑TTRIANGLE↑T
MAKE ↑T4 C↑T ↑TTRIANGLE↑T
MAKE ↑T5 C↑T ↑TTRIANGLE MARK I↑T
MAKE ↑T6 C↑T ↑TTRIANGLE↑T
MAKE ↑T7 C↑T ↑TTRIANGLE↑T
MAKE ↑T8 D↑T ↑TNOT-CURRENT↑T
MAKE ↑TSTRUC MARK↑T ↑T9 0 10 11 12 13 160 ERASE↑T
MAKE ↑T10 C↑T ↑TMARK↑T
MAKE ↑T11 C↑T ↑TMARK↑T
MAKE ↑T12 C↑T ↑TMARK↑T
MAKE ↑T13 C↑T ↑TMARK MARK↑T
MAKE ↑T14 C↑T ↑TMARK↑T
MAKE ↑TSTRUC SUPERMARK↑T ↑T15 0 16 17 160 ERASE↑T
MAKE ↑T16 C↑T ↑TSUPERMARK MARK↑T
MAKE ↑T17 C↑T ↑TSUPERMARK MARK↑T
MAKE ↑T18 C↑T ↑TSUPERMARK↑T
MAKE ↑TSTRUC RECTANGLE↑T ↑T19 0 20 21 22 23 24 160 ERASE↑T
MAKE ↑T20 C↑T ↑TRECTANGLE SUPERMARK↑T
MAKE ↑T21 C↑T ↑TRECTANGLE↑T
MAKE ↑T22 C↑T ↑TRECTANGLE↑T
MAKE ↑T23 C↑T ↑TRECTANGLE↑T
MAKE ↑T24 C↑T ↑TRECTANGLE RECTANGLE↑T
MAKE ↑T25 C↑T ↑TRECTANGLE↑T
MAKE ↑TSTRUC SUBRECTANGLE↑T ↑T26 0 27 28 29 30 31 160 ERASE↑T
MAKE ↑T27 C↑T ↑TSUBRECTANGLE SUPERMARK↑T
MAKE ↑T28 C↑T ↑TSUBRECTANGLE↑T
MAKE ↑T29 C↑T ↑TSUBRECTANGLE↑T
MAKE ↑T30 C↑T ↑TSUBRECTANGLE↑T
MAKE ↑T31 C↑T ↑TSUBRECTANGLE RECTANGLE↑T
MAKE ↑T32 C↑T ↑TSUBRECTANGLE↑T
MAKE ↑TSTRUC FIND↑T ↑T33 0 34 35 36 37 160 ERASE↑T
MAKE ↑T34 C↑T ↑TFIND↑T
MAKE ↑T35 C↑T ↑TFIND↑T
MAKE ↑T36 C↑T ↑TFIND↑T
MAKE ↑T37 C↑T ↑TFIND FIND↑T
MAKE ↑T38 C↑T ↑TFIND↑T
MAKE ↑TSTRUC HALF↑T ↑T39 0 40 41 160 ERASE↑T
MAKE ↑T40 C↑T ↑THALF↑T
MAKE ↑T41 C↑T ↑THALF FIND↑T
MAKE ↑T42 C↑T ↑THALF↑T
MAKE ↑TSTRUC DELETE↑T ↑T43 0 44 45 46 160 ERASE↑T
MAKE ↑T44 C↑T ↑TDELETE↑T
MAKE ↑T45 C↑T ↑TDELETE↑T
MAKE ↑T46 C↑T ↑TDELETE DELETE↑T
MAKE ↑T47 C↑T ↑TDELETE↑T
MAKE ↑TSTRUC STRIPE↑T ↑T48 0 49 50 51 52 53 160 ERASE↑T
MAKE ↑T49 C↑T ↑TSTRIPE SUBRECTANGLE↑T
MAKE ↑T50 C↑T ↑TSTRIPE SUBRECTANGLE↑T
MAKE ↑T51 C↑T ↑TSTRIPE↑T
MAKE ↑T52 C↑T ↑TSTRIPE↑T
MAKE ↑T53 C↑T ↑TSTRIPE STRIPE↑T
MAKE ↑T54 C↑T ↑TSTRIPE↑T
MAKE ↑TSTRUC MIDDLE↑T ↑T55 0 56 57 160 ERASE↑T
MAKE ↑T56 C↑T ↑TMIDDLE MARK HALF HALF↑T
MAKE ↑T57 C↑T ↑TMIDDLE↑T
MAKE ↑T58 C↑T ↑TMIDDLE↑T
MAKE ↑T60 D↑T ↑TALREADY-DEFINED↑T
MAKE ↑T61 C↑T ↑TTRIANGLE↑T
MAKE ↑T63 C↑T ↑TTRIANGLE MARK↑T
MAKE ↑T64 C↑T ↑TTRIANGLE MARK↑T
MAKE ↑T65 C↑T ↑TTRIANGLE↑T
MAKE ↑T66 C↑T ↑TTRIANGLE↑T
MAKE ↑T67 C↑T ↑TTRIANGLE↑T
MAKE ↑T69 C↑T ↑TTRIANGLE↑T
MAKE ↑T70 C↑T ↑TTRIANGLE MARK↑T
MAKE ↑T71 C↑T ↑TTRIANGLE↑T
MAKE ↑T72 C↑T ↑TTRIANGLE↑T
MAKE ↑T73 C↑T ↑TTRIANGLE↑T
MAKE ↑T74 C↑T ↑TSUBRECTANGLE↑T
MAKE ↑T75 C↑T ↑TSUPERMARK↑T
```

TABLE 4. Listing of a $PARSE Output

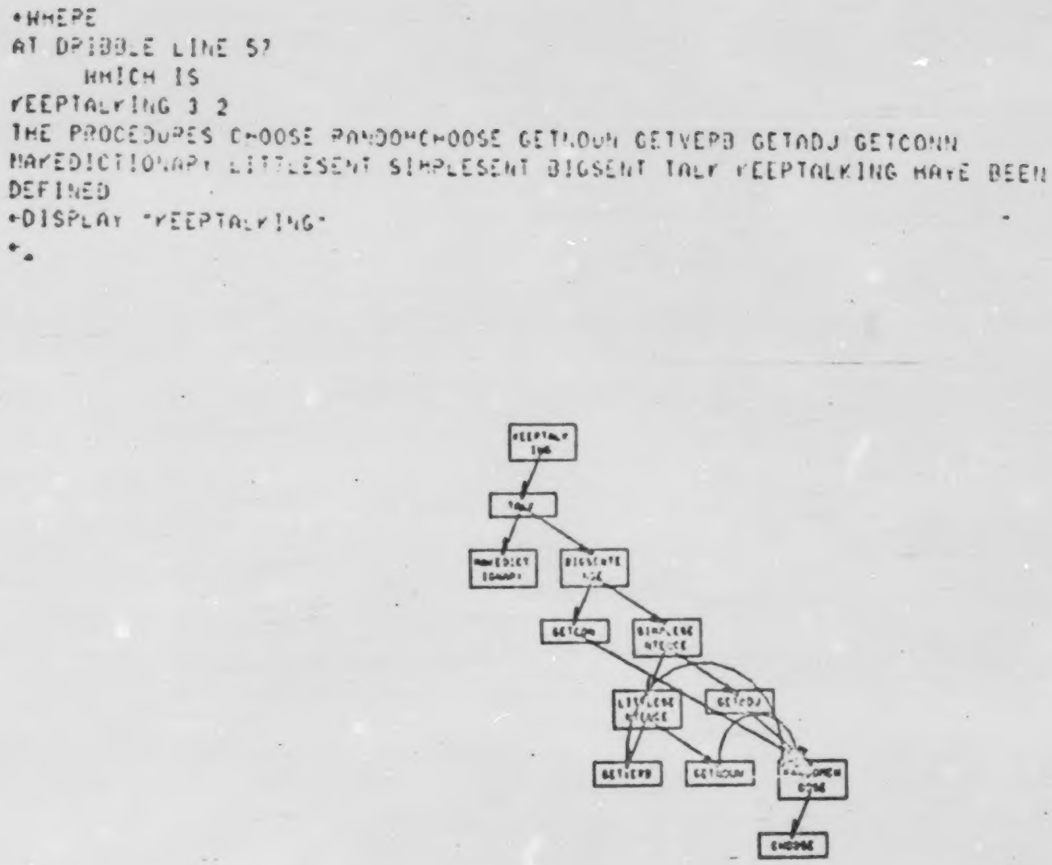
9. RUNning the Dribble File

In this section we describe the RUN system and the RUN phase of the analysis. Once the parsing of the dribble file, with its attendant generation of auxiliary information, is completed, the file is ready to be studied. We call the analysis subsystem which is used for this the RUN system.

9.1 The Display Configuration

In its present form the RUN system uses an IMLAC display terminal for graphic as well as alphanumeric presentations. The specific graphics commands and their implementation are discussed in the system documentation (Part 2); here it suffices to mention that these include both relative and absolute vector drawing capabilities as well as various alphabetic subroutines. The graphics capability is chiefly the drawing of procedure structure diagrams similar to those shown in Figures 1 and 2 above; the bulk of the information presented is in alphanumeric form -- this includes both the analyst's commands to the system and the unfolding of the dribble file, i.e., the student commands and their consequent output.

The alphanumerics are kept separate from the graphics -- the top part of the IMLAC screen serves as a scrolled teleprinter, the lower helf serving for graphics. A sample screen is shown in Figure 3. (The photocopy was made by a hard-copy device associated with the IMLAC.) In the situation shown, the dribble file user typed WHERE to the dribble file system which responded immediately below, telling him what point has been reached in the analysis. The user then typed a DISPLAY command (DISPLAY "KEEPTALKING") which resulted in the procedure structure diagram being displayed



```
←WHERE
AT DRIBBLE LINE 57
     WHICH IS
KEEPTALKING 3 2
THE PROCEDURES CHOOSE RANDOMCHOOSE GETNOUN GETVERB GETADJ GETCONN
MAKEDICTIONARY LITTLESENT SIMPLESENT BIGSENT TALK KEEPTALKING HAVE BEEN
DEFINED
←DISPLAY "KEEPTALKING"
←
```

(Note: The procedure names in the boxes, although quite readable on the screen, do not reproduce legibly.)

Figure 3. Example of a RUN system Interaction on the IMLAC Display Terminal.

underneath. Procedure structure diagrams are discussed below, as are WHERE, DISPLAY, and the other RUN system commands.

9.2 The Augmented Dribble File

In RUN phase, the analyst has a considerably augmented dribble file to work with. In addition to a copy of the original file, he has the auxiliary data summarized in Table 3 above. Generally speaking, these fall into two classes, the automatically generated "comments" "(n) A" "(n) B" and "(n) D" and the reconstruction aids of the form "(n) C", "FINAL CONTENTS", "STRUC (pname)". In addition to providing textual information, these comments may be considered a descriptor set that can be used as switches by run-time commands. A standard example of the use of such a switch is in the conditional execution of the lines of a dribble file, as when the analyst skips over those obviously defective lines labeled by a non-empty "n (D)".

Besides the special, temporarily used variables generated by the graphics package subroutines, there are just two global variables which are monitored by the RUN system. "DRIBBLE NO", initially 0, gives the line no. up to which the dribble file has been executed. "CURRENT PROCEDURE", initially empty, gives the name of the procedure currently being defined in the course of execution of the dribble file. (The latter piece of information, although obviously known to the LOGO language system, is not otherwise accessible to LOGO programs.)

9.3 Execution of Dribble File Lines

To begin use of RUN, the command $STARTRUN asks for a dribble file name and initializes the system. The analyst then has two essentially distinct modes of interaction. First, and most straightforward, he can execute the dribble file sequentially in a variety of different ways, depending upon the kind of information he wishes to extract. Alternatively, he can interrogate the dribble file without any execution, possibly in a nonsequential manner. And, of course, he can intermix these two modes.

In the first mode of work, the analyst has three kinds of execution facilities available to him. First, he can simply look at the lines of the dribble file as they are being executed together with the computer responses that they generate. He can execute the dribble file one line at a time using the command $DOLINE. This results in the next dribble file line being printed (with three asterisks on each side to set it off) and then the execution of the line, if it has parsed correctly. $DOLINE then stops. He can perform a specified number of lines with the command $DOTO :LINE NUMBER:. And, he has available a conditional stop as follows. $DOUNTIL :DESCRIPTOR: executes the dribble file until the specified :DESCRIPTOR: is found (as the comment associated with that dribble file line). Finally, $DOALL executes the entire dribble file line-by-line. $DOALL, $DOTO, and $DOUNTIL call $DOLINE repeatedly.

A second source of information is from "looking around," after having executed the dribble file up to some point. The analyst can list the definition of any procedure in the dribble file which is defined at that point. He can execute any procedure (the student's or his own) or, generally speaking, perform any LOGO commands. There is, of course, the danger that in certain activities, such as redefining student procedures, the user will destroy the "state of the world." Therefore, we have provided the self-explanatory commands $SAVEWORLD and $GETWORLD to enable a user to save the state of the world, make any modifications that he might desire, and when he is finished to get back to where he was. It is, of course, only necessary to use these commands when the user intends to type LOGO commands which will modify the existing procedure structures or global names. The user may define his own LOGO procedures to aid in the "looking around" process. The Analysis Package (Part 3) contains several examples of debugging and modification routines which are typical of such procedures. (The naming conventions -- sentences for global procedure names and $ as first character of procedure name must, of course, be adhered to in such procedures to avoid conflict with student programs.)

The third execution-based type of information is a diagram of the procedure structure at any point in the dribble file. By using the command $DISPLAY :ROOT: the user generates a graphic representation of the procedure structure starting at :ROOT:. The primitives for this display are generated as part of the augmentation of the dribble file during "parsing."

$WHERE is a general interrogatory command which tells the user where he is in the dribble file--that is, up to which line he has executed, and the set of procedures which have been defined at this point. As part of the graphics display, if a procedure is being defined at that point, we also show, in smaller print, the entire definition of the procedure at that moment in time.

For the other major mode of run-time analysis, a completely separate class of commands enables the user to interrogate the dribble file without any execution. The command $ALLDESCR outputs a list of all the different descriptors used in comments throughout the dribble file. This command is useful in connection with subsequent execution because any of these descriptors can be used as a modifier with $DOUNTIL to terminate execution. We can also obtain the set of lines containing any given descriptor with the command $FINDLINES :LINE NUMBER: :DESCRIPTOR:. This command types all the lines after :LINE NUMBER: in the dribble file which contain that descriptor.

We can readily construct the definition of any procedure at any point in the dribble file using only information that was there during parsing, i.e., without any execution whatever. $STEPTHROUGH :PROCEDURE NAME: prints the successive variations of :PROCEDURE NAME: as subsequent lines were defined: it first gives the original title of the procedure, then the first line, etc. If a line is replaced, the new line replaces the old line. Thus the definition of :PROCEDURE NAME: "unfolds" in time.

9.4 Modifying the Display

Two predicate procedures are included in the package to facilitate user modification of display and execution of dribble file lines. (Of course, an ambitious user can modify the package in other ways.) $NICEP enables the user to define additional criteria for execution or non-execution of dribble file lines. (Lines which parse badly are never executed independent of $NICEP.) If $NICEP outputs "FALSE", the line is not executed. Initially, $NICEP is defined as,

```
TO $NICEP
10 OUTPUT "TRUE"
END
```

Additional lines of the form IF OUTPUT "FALSE" can be added to achieve the desired effect.

$SHOWLINEP can be used to control the form of the dribble line display. For example, the user may not wish to see the stored lines being retrieved by a GET contained in the file. Initially $SHOWLINEP is defined as:

```
TO $SHOWLINEP
10 OUTPUT "FALSE"
END
```

This defaults to the standard display of each line. The output "TRUE" overrides this display, thus to avoid seeing stored lines, for example, merely change line 10 to

```
10 IF $STOREDP :DRIBBLE NO: OUTPUT "TRUE"
    ELSE OUTPUT "FALSE"
```

where $STOREDP merely reports the non-emptiness of "(:DRIBBLE NO:) A".

9.5 Summary of RUN Commands

The RUN system commands are listed and briefly described in Table 5 following.

$STARTRUN	initializes RUN system
$DOLINE	executes next line of dribble file
$DOTO :N:	executes until line :N:
$DOUNTIL :COND:	executes until the comment :COND: is found
$DOALL	executes entire dribble file
$SAVEWORLD	save current status of workspace
$GETWORLD	retrieve previously saved status
$WHERE	prints information on current position in file
$DISPLAY :ROOT:	displays procedure structure as of current position in dribble file, starting at :ROOT:
$ALLDESCR	lists all comments in dribble file
$FINDLINES :N: :DESCRIPTOR:	lists all lines after :N: containing :DESCRIPTOR: as part of the comment
$STEPPROCEDURE :PNAME:	"unfolds" definition of :PNAME: across entire dribble file
$NICEP $SHOWLINEP	user-defined procedures to modify display

TABLE 5. Summary of RUN Commands.

Part 2.

System Documentation

1. Introduction to System Documentation

In the preceding User's Guide (Part 1) we have given a general description of the parts of our dribble file analysis system as well as instructions on their use. Here we present the annotated set of programs as a more complete documentation of our system. For ease in user extension and customization, all these programs except for a few very trivial front end procedures were written in the LOGO programming language, a complete description of which is contained in the Appendix.

Generally speaking, there are two types of information of interest to a user in the analysis of dribble files. First there are queries which involve, or potentially involve, the entire state of the world created by the student's work up to that point. (What happens when this program is run with inputs X and Y?) A common query of this kind calls for execution of a procedure, possibly invoking any number of other procedures.

On the other hand, a question may require only a very small local part of the state of the world for its answer: at what point was procedure P defined? what was its definition at time t?, etc. Questions involving the entire state of the world require that the entire state of the world be recreated. It turns out, however, that auxiliary information generated during pre-processing of the dribble file can be used to directly answer broad classes of this latter type of question without having difficult searches or any live execution.

Furthermore, as we will describe later, the preprocessing can be modified by user-written or user-specified programs so that auxiliary information is generated in advance to answer efficiently whatever additional classes of "local" questions are deemed to be of interest.

Thus, we subdivide the preprocessing phase into two subphases, first, the preprocessing proper in which relatively straightforward modifications are made to the text to put it into a form better suited for further work and, second, a preanalysis or parsing phase in which the auxiliary information mentioned above is generated. These two parts of the preprocessing are sharply distinguished in our implementation, which has a TECO macro preprocessor driving a LOGO "parsing analyzer."

Section 2 of the system documentation contains a brief description of the ground rules under which the programming was performed. Sections 3, 4, and 5 contain annotated listings of the three parts of the system described above -- preprocessing, parsing, and running. Further information on the graphics capabilities utilized is required, since LOGO graphics tend to be relatively system-dependent. Section 6 contains a description both of the facilities used and of other possible configurations.

2. The Use of LOGO as Both System and Object Language

In addition to using the LOGO programming language as an "object language" for student programming work, we have chosen it as the vehicle for implementation of the dribble file analysis system. Our choice of LOGO for the analysis system programming language was made with considerable deliberation after considering several other alternatives, including LISP, PL-I, and FORTRAN IV. Those very features which make LOGO so useful in the classroom are the ones that enable a relatively unsophisticated user to modify and extend the programs in the dribble file analysis system. Chief among these features are a procedural-based programming heuristic and an unusually pure and context-free syntax.

We realized that there are certain dangers in choosing the same language for both the analysis and the object being analyzed. The fact that, in a sense, LOGO syntax is used to analyze LOGO syntax can lead to a system inbred to a degree which makes transferability to other object languages impossible in any practical sense. For example, error diagnostics can be generated directly from the run-time stacks instead of by an external parsing procedure. In the extreme case, the dribble file analysis system could become just an extension of the LOGO interpreter.

To avoid such difficulties, we made the firm rule that the only way in which the LOGO interpreter would be used in conjunction with the contents of dribble files would be via an EXECUTE LINE type of command (that is, in "running" dribble files. All sorts of editing and viewing are permitted as long as they are external to the LOGO evaluator.). Everything else would be done by procedures external to this interaction (which "happen" to be written in LOGO). This decision enables the system we have developed to be useable with any language having some sort of execute command (i.e., with virtually all languages). All that needs to be done is to have the execute command implemented inside the LOGO interpreter. More will be written later about differences between various "object languages" vis-a-vis the design and use of the dribble file analysis system.

3. The Preprocessing

A general description of preprocessing appears in the User's Guide. The preprocessing is done by a set of eight TECO* macros, which are called in turn by a "loading" macro that also elicits input and output file name. The specific effects of each macro are as follows:

LOADER) Loads rest of TECO macros, elicits input and output file designations, and executes macros.

1) Erases first line of dribble file which just contains identification. Changes CR and ↑S to EOL (ASCII 31) to ensure proper line termination. Checks for last character of file being EOL . Changes line feed to space.
2) Finds rubouts, deletes line to that point. Marks RUBOUT at end of preceding line.
3) Finds lines containing ↑N and ↑R and changes all editing characters to textual equivalents, since at this point it would be too difficult to recover the context in which ↑N and ↑R operate.
4) Performs effects of \ and ↑W , in order, left to right.
5) Finds stored lines and marks them with @.
6) Kills time marks.
7) Numbers lines and comments (RUBOUT and STORED) in form as LOGO names.
8) Spaces out separators.

---

*TECO is the character-oriented text editing language on the DEC PDP-10. It is described in the DEC System-10 User's Handbook and the modifications made by BBN are given in TENEX TECO, a BBN report, NIC 19937.

LOADER.TEC

```
◆HT$
ERONE.TEC$Y$HX1$ERTWO.TEC$HX2$ERTHREE.TEC$HX3$ERFOUR.TEC$HX4$ERFIVE.TEC$
◆◆HX5$ERSIX.TEC$HX6$ERSEVEN.TEC$HX7$EREIGHT.TEC$HX8$;Y$M1$M2$M3$M4$M5$M6
◆◆$M7$M8$EX$
◆
```

ONE.TEC

CR

LINEFEED

```
$
0J$K$<:S$; $-D$31I$>0J$<:S
                                        $; -D$I $>$<:S^S$; $-D$31I$>$(Z-1)J$(1A-31)"N$
◆◆31I$'
```

TWO.TEC

RUBOUT

```
$
0J$<:S$; $0K$-2C$(1A-127)"N$127I$'$>
◆
```

THREE.TEC

```
$
0J$<:S^N$; 0K<(1-((1A-31)◆(1A-31))); (1A-92)"E$-D$I%%SLASH$'$(1A-23)"E$-
◆◆D$ICTRL-W$'$(1A-14)"E$-D$ICTRL-N$'$(1A-13)"E$-D$ICTRL-R$'$>$>0J$<:S$;
◆◆0K<(1-((1A-31)◆(1A-31))); (1A-12)"E$-D$I%%SLASH$'$(1A-23)"E$-D$ICTRL-W
◆◆$'$(1A-14)"E$-D$ICTRL-N$'$(1A-13)"E$-D$ICTRL-R$'$>$>
◆
```

FOUR.TEC

```
$
0J$!BEG!C$(.-Z)"E$OEND$'$(1A-92)"E$-2D$OBEG$'$(1A-23)"E$-D$<-C$(1-((1A-3
◆◆2)◆(1A-32)◆(1A-31)◆(1A-31))); $D$>$'$OBEG$!END!$
◆
```

FIVE.TEC

```
$
0J$<:S0:00:00$; L-C$I?$>0J<:S24:00:00$; -D$L-C$I?$>
◆
```

SIX.TEC

```
$
0J<(.-Z+1); 3D$!LOOP!$(1A-32)"E$D$OLOOP$'$(1A-127)"E$D$OLOOP$'$(1A-64)"E
◆◆$D$OLOOP$'$(1A-31)"E$-D$'$:S
$; $>
◆
```

SEVEN.TEC

```
$
0J$0UZ$<(.-Z+1); IMAKE$20I$%Z\$I N$20II $20I$L-2C$(1A-64)"E$D0L$IMAKE$20
◆◆I$QZ\I A$20II $20II$ISTORED$20I$I
$L-2C$'$(1A-127)"E$D0L$IMAKE$20I$QZ\$I B$20I$I $20IIRUBOUT$20I$I
$L-2C$'$C$20I$C$>
◆
```

EIGHT.TEC

```
$
0J<:S\$;$-2D$>$0J<:S"$;$-1C$I $1C$I $>$0J<:S:$;$-1C$I $1C$I $>0J<:S;$;$-
◆◆1C$I $1C$I $>0J<:S+$;$-1C$I $1C$I $>0J<:S-$;$-1C$I $1C$I $>$0J<:S◆$;$-
◆◆1C$I $1C$I $>0J<:S/$;$-1C$I $1C$I $>0J<:S<$;$-1C$I $1C$I $>0J<:S>$;$-1
◆◆C$I $1C$I $>$0J<:S=$;$-1C$I $1C$I $>$
◆
```

4. Parsing

The procedure structure of this section of the program is quite completely discussed in the User's Guide. The main omission there is discussion of multiline commands - EDT, EDL, and two-line MAKE commands. These are all handled correctly by the parser, except in those cases for the first two in which ↑N and ↑R are used. The parser simply puts such lines as comments following the EDT or EDL instead of expanding them. It is not particularly difficult to handle this properly if it is felt necessary to do so. The annotated program listings follow. The main procedures are given in a fairly natural order, followed by the small utility programs marked U, and empty user-definable procedures (as described in the User's Guide, Part 1) marked E.

[1]

```
TO $PARSE ;TOP LEVEL PARSING PROCEDURE;
 0 $$INITNAMES
20 TYPE "DRIBBLE FILE:"
30 DO SENTENCE "GET" REQUEST
40 $GOTHROUGH 1
END
```

[2]

```
TO $$INITNAMES ;INITIALIZES LIST OF BUILTINS TO BE OF FORM "(BUILTIN) BP
                                             **" IS "A";
 0 ERASE ALL NAMES
20 $$IX $$BP
END
```

[3]

```
TO $$IX :LIST: ;USED BY INITNAMES;
 0 IF EMPTYP :LIST: STOP
20 MAKE SENTENCE "BP" FIRST :LIST: "A"
30 $$IX BUTFIRST :LIST:
END
```

[4]

```
TO $$BP ;USED BY $$IX;
 0 OUTPUT "ABBREVIATE AND ASK AS BACK BOTH BUTFIRST BUTLAST CANCEL CLOCK
                                    ** COUNT DATE DIFFERENCE DIVISION DO
                                    ** EDIT EDITLINE EDITTITLE EITHER EM
                                    **PTYP END ENTRIES ERASE ERASELINE E
                                    **XIT FIRST FRONT GET GO GOTOLINE GO
                                    **ODBYE GREATERP HORN IFFALSE IFTRUE
                                    ** IGNORE IS LAST LEFT LINES LIST LO
                                    **CAL MAKE MAXIMUM MINIMUM NUMBERP O
                                    **F OUTPUT PRINT PRODUCT QUOTIENT RA
                                    **NDOM REMAINDER REQUEST RESETCLOCK
                                    **RIGHT SAVE SENTENCE SENTENCEP SIZE
                                    ** STOP SUM TEXT TEST THING TIME TIT
                                    **LE TO TOUCHLEFT TOUCHRIGHT TRACE T
                                    **YPE WAIT WORD WORDP ZEROP + - * /
                                    **) ( IF THEN ELSE ABB BF BL C DIFF
                                    **EDL EDT EI EP ER ERL F GTL GB GP I
                                    **FF IFT L MAX MIN NP OP P PR QUO RE
                                    **M REQ S SP SS T WP W ZP SENTENCES
                                    **SS ="
END
```

[5]

```
TO $GOTHROUGH :DRIBBLE NO: ;ITERATES PARSING THROUGH LINES;
5 MAKE "CURRENT LINE" SENTENCES THING SENTENCE :DRIBBLE NO: "N"
10 TEST EMPTYP :CURRENT LINE:
20 IFTRUE $CLEANUP
30 IFTRUE STOP
40 $STARTLINE
50 $GOTHROUGH SUM :DRIBBLE NO: 1
END
```

[6]

```
TO $STARTLINE ;PARSES ONE LINE, DISPATCHES ON TYPE OF LINE TO $DOLINE,$2
                                    **MAKE (2LINE MAKE) $STORED OR ONE O
                                    **F THE PROCEDURES INDICATED ON LINE
                                    ** 110;
 0 $EXAMINELINE
20 IF $USEUPLINE STOP
30 MAKE "CURRENT LINE" $SUCKALLSEMIS :CURRENT LINE:
40 TEST $STOREDP
50 IFTRUE $ENDLINE
60 IFTRUE STOP
70 TEST $MP FIRST :CURRENT LINE: "GET SAVE GOODBYE GB CANCEL GO"
80 IFTRUE $ADDERROR "NOT-CURRENT"
90 IFTRUE $ENDLINE
 00 IFTRUE STOP
 10 IF NOT EITHER $2MAKE $DISPATCHP FIRST :CURRENT LINE: "TO $PARSETO ED
                                    **IT $PARSEEDIT END $PARSEEND ERASE
                                    **$PARSEERASE ERL $PLERASE EDL $PLED
                                    **IT EDT $PTEDIT TITLE $PTITLE" THEN
                                    ** $DOLINE

 20 $ENDLINE
END
```

[7]

```
TO $DISPATCHP :EL: :LIST: ;CALLS PROCEDURE ON :LIST: WHICH FOLLOWS :EL:,
                                    ** OUTPUTS "FALSE" ON FAILURE;
 0 IF EMPTYP :LIST: OUTPUT "FALSE"
20 TEST IS :EL: FIRST :LIST:
25 IFTRUE MAKE "CURRENT LINE" BUTFIRST :CURRENT LINE:
30 IFTRUE DO FIRST BUTFIRST :LIST:
40 IFTRUE OUTPUT "TRUE"
50 OUTPUT $DISPATCHP :EL: BUTFIRST BUTFIRST :LIST:
END
```

[8]

```
TO $DOLINE ;HANDLES "NONSPECIAL" LINE;
10 IF EMPTYP :CURRENT LINE: STOP
20 $EXAMINEL
22 TEST $U[illegible]UPEL
25 IFTRUE $DOLINE
28 IFTRUE STOP
30 TEST EITHER NUMBERP FIRST :CURRENT LINE: $BUILTINP FIRST :CURRENT LIN
                                        **E:
40 IFTRUE MAKE "CURRENT LINE" BUTFIRST :CURRENT LINE:
50 IFTRUE $DOLINE
60 IFTRUE STOP
70 TEST IS FIRST :CURRENT LINE: :QUOTE:
80 IFTRUE MAKE "CURRENT LINE" $SUCKQUOTE BUTFIRST :CURRENT LINE:
90 IFTRUE $DOLINE
100 IFTRUE STOP
110 TEST IS FIRST :CURRENT LINE: ";"
120 IFTRUE MAKE "CURRENT LINE" $SUCKSEMI BUTFIRST :CURRENT LINE:
130 IFTRUE $DOLINE
140 IFTRUE STOP
150 TEST IS FIRST :CURRENT LINE: ":"
160 IFTRUE MAKE "CURRENT LINE" $SUCKDOTS BUTFIRST :CURRENT LINE:
170 IFTRUE $DOLINE
180 IFTRUE STOP
190 $ADDSTRUC FIRST :CURRENT LINE:
200 MAKE "CURRENT LINE" BUTFIRST :CURRENT LINE:
210 $DOLINE
END
```

[9]

```
TO $2MAKE ;HANDLES 2 LINE MAKES;
10 IF NOT IS :CURRENT LINE: "MAKE" OUTPUT "FALSE"
20 $PL2MAKE
30 OUTPUT "TRUE"
END
```

[10]

```
TO $PL2MAKE ;USED BY $2MAKE;
10 IFTRUE $ADDERROR "MULTIMAKE-INTERRUPTED"
20 IFTRUE $ADDERROR "MULTI-MAKE-INTERRUPTED"
30 IFTRUE STOP
40 TEST $RUBOUTP SUM :DRIBBLE NO: 1
50 IFTRUE $ADDERROR "MULTI-MAKE-INTERRUPTED"
60 IFTRUE $ADD SENTENCE SUM :DRIBBLE NO: 1 "D" "MULTI-MAKE-INTERRUPTED"
70 IFTRUE STOP
80 $ADD SENTENCE :DRIBBLE NO: "N" SENTENCE THING SUM :DRIBBLE NO: 1 "N"
                                       **THING SENTENCE SUM :DRIBBLE NO: 2
                                       **"N"
90 $ADD SENTENCE SUM :DRIBBLE NO: 1 "D" "IGNORE"
100 $ADD SENTENCE SUM :DRIBBLE NO; 2 "D" "IGNORE"
110 MAKE "CURRENT LINE" THING SENTENCE :DRIBBLE NO: "N"
120 $STARTLINE
END
```

[11]

```
TO $STOREDP ;USED BY $STARTLINE TO HANDLE PROCEDURE LINE;
10 IF NOT NUMBERP FIRST :CURRENT LINE: OUTPUT "FALSE"
20 $DOSTORED
30 OUTPUT "TRUE"
END
```

[12]

```
TO $DOSTORED ;USED BY STOREDP;
10 IF EMPTYP :CURRENT PROC: $ADDERROR "NOT-DEFINING"
20 IF GREATERP 1 FIRST :CURRENT LINE: $ADDERROR "INVALID-LINENO"
23 $ADDSTRUC SENTENCE :CURRENT PROC: FIRST :CURRENT LINE:
25 MAKE "CURRENT LINE" BUTFIRST :CURRENT LINE:
30 $DOLINE
40 IF $GOODPARSEP :DRIBBLE NO: $ADD SENTENCE "STRUC" :CURRENT PROC: SENT
                                       **ENCE :DRIBBLE NO: FIRST THING SENT
                                       **ENCE :DRIBBLE NO: "N"
50 IF NOT $GOODPARSEP :DRIBBLE NO; MAKE SENTENCE :DRIBBLE NO: "C" :EMPTY
                                       **:
END
```

[13]

```
TO $PARSETO ;CALLED BY $STARTLINE FOR PROCEDURE DEFINITIONS;
10 $DOTITLE FIRST :CURRENT LINE: BUTFIRST :CURRENT LINE:
END
```

[14]

```
TO $DOTITLE :NAME: :ARGLIST: ;USED BY $PARSETO FOR PROCEDURE DEFINITION
                                              **LINES;
10 $CHECKNAME :NAME:
50 $STARTDEF :NAME: $COUNTARGS :ARGLIST: 0
END
```

[15]

```
TO $CHECKNAME :NAME: ;CHECKS VALIDITY OF NAME FOR DEFINITION OR
                                              **REDEFINITION;
10 IF NUMBERP :NAME: $ADDERROR "NUMBER-NAME"
20 IF $MP :NAME: SENTENCE "( ) : ;" :QUOTE: $ADDERROR "SEPARATOR-NAME"
30 IF $BUILTINP :NAME: $ADDERROR "BUILTIN-NAME"
40 IF $DEFINEDP :NAME: $ADDERROR "ALREADY-DEFINED"
END
```

[16]

```
TO $STARTDEF :NAME: :N: ;PARSES TITLE LINE;
10 IF NOT $GOODPARSEP :DRIBBLE NO: STOP
15 MAKE "CURRENT PROC" :NAME:
20 $ADD SENTENCE "STRUC" :NAME: SENTENCE :DRIBBLE NO: 0
30 $ADD "FINAL CONTENTS" SENTENCES :NAME: :DRIBBLE NO: :N:
END
```

[17]

```
TO $COUNTARGS :LIST: :CTR: ;CHECKS VALIDITY AND COUNTS ARGS OF ARGLIST I
                                        **N TITLE;
5 IF EITHER IS FIRST :LIST: "AND" IS FIRST :LIST: "OF" OUTPUT $COUNTARGS
                                        **  BUTFIRST :LIST: :CTR:
10 IF EMPTYP :LIST: OUTPUT :CTR:
20 IF IS FIRST :LIST: ":" OUTPUT $COUNTARGS $SUCKDOTS BUTFIRST :LIST: SU
                                        **M :CTR: 1
30 IF IS FIRST :LIST: ";" OUTPUT $COUNTARGS $SUCKSEMI BUTFIRST :LIST: :C
                                        **TR:
40 $ADDERROR "BAD-ARGUMENT"
50 OUTPUT 0
END
```

[18]

```
TO $DEFINEDP :NAME: ;IS :NAME: CURRENTLY DEFINED?;
10 MAKE "NAME" THING SENTENCE "STRUC" :NAME:
20 IF EITHER EMPTYP :NAME: IS LAST :NAME: "ERASE" OUTPUT "FALSE"
30 OUTPUT "TRUE"
END
```

[19]

```
TO $PARSEEDIT ;USED BY $STARTLINE FOR ALL EDIT COMMANDS;
10 TEST IS FIRST :CURRENT LINE: "LINE"
20 IFTRUE $PLEDIT
30 IFTRUE STOP
33 TEST IS :CURRENT LINE: "TITLE"
36 IFTRUE $PTEDIT
38 IFTRUE STOP
40 TEST IS COUNT :CURRENT LINE: 1
50 IFFALSE $ADDERROR "EXTRA-ARG-IN-EDIT"
60 IFFALSE STOP
70 TEST BOTH $DEFINEDP FIRST :CURRENT LINE: EMPTYP :CURRENT PROC:
80 IFFALSE $ADDERROR "BADEDIT"
90 IFTRUE MAKE "CURRENT PROC" FIRST :CURRENT LINE:
END
```

```
   [20]

TO $PTEDIT ;TITLE EDIT-IGNORE THIS COMMAND AND PRECEDE NEXT LINE WITH "T
                                         **ITLE";
 0 TEST EMPTYP BUTFIRST :CURRENT LINE:
20 IFFALSE $ADDERROR "BAD-TITLE-EDIT"
30 IFFALSE STOP
40 $ADDERROR "IGNORE"
50 IF NOT $RUBOUTP :DRIBBLE NO: MAKE SENTENCE SUM :DRIBBLE NO: 1 "N" SEN
                                         **TENCE "TITLE" THING SENTENCE SUM :
                                         **DRIBBLE NO: 1 "N"
END

   [21]

TO $PLEDIT ;LINE EDIT-PUTS EDITTING COMMANDS ON LINE FOLLOWING AS COMMEN
                                         **TS AT END OF PRESENT LINE FOR HAND
                                         ** INSERTION BY USER;
 0 IGNORE $CHECKLINE
30 IF $RUBOUTP :DRIBBLE NO: STOP
40 $ADD SENTENCE ";" THING SENTENCE SUM :DRIBBLE NO: "N" SENTENCE :DRIBB
                                         **LE NO: "N"
50 $ADD SENTENCE SUM :DRIBBLE NO: 1 "D" "IGNORE"
END
   [22]

TO $PARSEEND ;USED BY $STARTLINE FOR THE END COMMAND;
 0 TEST EMPTYP :CURRENT LINE:
20 IFFALSE $ADDERROR "EXTRA-IN-END"
30 IFFALSE STOP
40 TEST EMPTYP :CURRENT PROC:
50 IFTRUE $ADDERROR "SUPERFLUOUS-END"
60 MAKE "CURRENT PROC" :EMPTY:
END

   [23]

TO $PARSEERASE ;USED BY $STARTLINE FOR ALL ERASE COMMANDS;
 0 TEST IS FIRST :CURRENT LINE: "LINE"
20 IFTRUE $PLERASE
30 IFTRUE STOP
33 TEST IS FIRST :CURRENT LINE: "ALL"
36 IFTRUE $PARSEERAL
38 IFTRUE STOP
40 TEST IS COUNT :CURRENT LINE: 1
50 IFFALSE $ADDERROR "BAD-ERASE"
60 IFFALSE STOP
70 TEST $DEFINEDP FIRST :CURRENT LINE:
80 IFFALSE $ADDERROR "NOTHING-TO-ERASE"
90 IFFALSE STOP
 00 IF IS "CURRENT PROC" FIRST :CURRENT LINE: MAKE "CURRENT PROC" :EMPTY
                                         **:
 10 $ERASEPR :CURRENT LINE:
END
```

[24]

```
TO $PLERASE ;HANDLES ERASE LINE;
 0 IF NOT $CHECKLINE STOP
40 IF $GOODPARSEP :DRIBBLE NO: $ADD SENTENCE "STRUC" :CURRENT PROC: SENT
                                    **ENCE :DRIBBLE NO: WORD "-" LAST :C
                                    **URRENT LINE:
END
```

[25]

```
TO $CHECKLINE ;CHECKS VALIDITY OF LINE BEING CALLED FOR ALTERATION OR ER
                                    **ASURE;
 0 TEST BOTH NUMBERP FIRST BUTFIRST ;CURRENT LINE: EMPTYP BUTFIRST BUTFI
                                    **RST :CURRENT LINE:
20 IFTRUE $ADDERROR "NOT-DEFINING"
25 IFTRUE OUTPUT "FALSE"
30 TEST EMPTYP :CURRENT PROCEDURE:
40 IFTRUE $ADDERROR "NOT-DEFINING"
50 IFTRUE OUTPUT "FALSE"
60 TEST $MP LAST :CURRENT LINE: THING SENTENCE "STRUC" :CURRENT PROC:
70 IFTRUE $ADDERROR "NO-SUCH-LINE"
80 IFTRUE OUTPUT "FALSE"
90 OUTPUT "TRUE"
END
```

[26]

```
TO $PARSEERAL ;HANDLES ALL ERASE ALLS;
 0 MAKE "CURRENT LINE" BUTFIRST :CURRENT LINE:
20 TEST EITHER EMPTYP ;CURRENT LINE: EITHER IS :CURRENT LINE: "PRS" IS :
                                    **CURRENT LINE: "PROCEDURES"
30 IFTRUE MAKE "CURRENT PROC" :EMPTY:
35 IFTRUE MAKE SENTENCE :DRIBBLE NO: "N" "$ERASEALL"
40 IFTRUE $PARSEERALL :FINAL CONTENTS:
45 IFTRUE STOP
50 IF IS :CURRENT LINE: "TRACES" STOP
60 TEST IS :CURRENT LINE: "NAMES"
70 IFTRUE $ADDERROR "NOT-CURRENT"
80 IFTRUE STOP
90 $ADDERROR "BAD-ERASE-ALL"
END
```

[27]

```
TO $PARSEERALL :LIST: ;HANDLES ERASE ALL;
 0 IF EMPTYP :LIST: STOP
20 IF $DEFINEDP FIRST :LIST: $ERASEPR FIRST :LIST:
30 $PARSEERALL BUTFIRST BUTFIRST BUTFIRST :LIST:
END
```

[28]

```
TO $ERASEPR :NAME: ;HANDLES ERASE :NAME:;
10 $ADD SENTENCE "STRUC" :NAME: SENTENCE :DRIBBLE NO: "ERASE"
20 $ADD "FINAL CONTENTS" SENTENCES ;NAME: :DRIBBLE NO: "ERASE"
END
```

[29]

```
TO $PTITLE ;HANDLES TITLE COMMAND;
10 TEST IS FIRST :CURRENT LINE: "TO"
20 IFTRUE $CHANGETITLE FIRST BUTFIRST :CURRENT LINE: BUTFIRST BUTFIRST :
                                         **CURRENT LINE:
30 IFFALSE $ADDERROR "BAD-TITLE-COMMAND"
END
```

[30]

```
TO $CHANGETITLE :NAME: ;DOES WORK FOR $PTITLE;
10 TEST IS :NAME: :CURRENT PROCEDURE:
20 IFFALSE $CHECKNAME :NAME:
30 IFFALSE TEST NOT $GOODPARSEP :DRIBBLE NO:
40 IFFALSE $ADD SENTENCE "STRUC" ;NAME: THING SENTENCE "STRUC" :CURRENT
                                         **PROC:
50 $STARTDEF :NAME: $COUNTARGS :ARGLIST: 0
END
```

[31]

```
TO $CLEANUP ;CLEANS UP AT END OF PARSING;
10 PRINT "PARSING COMPLETED, SAVE THE DRIBBLE FILE."
20 MAKE "DRIBBLE NO" 0
30 DO "ERASE ALL PROCEDURES"
END
```

UTILITY PROCEDURES

```
TO $SUCKALLSEMIS :LINE: ;REMOVES ALL PAIRS OF SEMICOLONS AND THEIR CONTE
                                         **NTS FROM :LINE:;
10 IF NOT IS FIRST :LINE: ";" OUTPUT :LINE:
20 OUTPUT $SUCKALLSEMIS $SUCKSEMI BUTFIRST :LINE:
END
```

```
TO $SUCKSEMI :LINE: ;UTILITY, SUCKS UP CONTENTS OF SEMICOLONS;
10 IF EITHER EMPTYP :LINE: IS FIRST :LINE: ";" OUTPUT BUTFIRST :LINE:
20 OUTPUT $SUCKSEMI BUTFIRST :LINE:
END


TO $BUILTINP :NAME: ;UTILITY, IS :NAME: A BUILTIN?;
10 IF EMPTYP THING SENTENCE "BP" :NAME: OUTPUT "FALSE"
20 OUTPUT "TRUE"
END

TO $SUCKQUOTE :LINE: ;UTILITY, SUCKS UP CONTENTS OF A PAIR OF QUOTES;
10 TEST EMPTYP :LINE:
20 IFTRUE $ADDERROR "MATCHING-QUOTE"
30 IFTRUE OUTPUT :EMPTY:
40 IF NOT IS FIRST :LINE: :QUOTE: OUTPUT $SUCKQUOTE BUTFIRST :LINE:
50 OUTPUT BUTFIRST :LINE:
END

TO $SUCKDOTS :LINE: ;UTILITY, SUCKS UP CONTENTS OF A PAIR OF COLONS;
10 TEST IS FIRST :LINE: ":"
20 IFTRUE $ADDERROR "EMPTY-NAME"
30 IFTRUE OUTPUT :EMPTY:
40 OUTPUT $SUCKDOTS1 BUTFIRST :LINE:
END

TO $ADDSTRUC :EL: ;UTILITY, ADDS :EL: TO SET OF STRUCTURE ELEMENTS BELON
                                         **GING TO "(DRIBBLE NO) C";
10 IF NOT EMPTYP :CURRENT PROC: MAKE SENTENCE :DRIBBLE NO: "C" SENTENCE
                                         **THING SENTENCE :DRIBBLE NO: "C" :E
                                         **L:
END

TO $GOODPARSEP :N: ;UTILITY, DOES DRIBBLE LINE :N: HAVE ANY PARSING ERRO
                                         **RS?;
10 OUTPUT EMPTYP THING SENTENCE :N: "D"
END

TO $SUCKDOTS1 :LINE: ;USED BY SUCKDOTS;
10 TEST EMPTYP :LINE:
20 IFTRUE $ADDERROR "MATCHING-DOTS"
30 IFTRUE OUTPUT :EMPTY:
40 IF NOT IS FIRST :LINE: ":" OUTPUT $SUCKDOTS1 BUTFIRST :LINE:
50 OUTPUT BUTFIRST :LINE:
END
```

```
TO $MP :THING: :LIST: ;UTILITY, IS :THING: AN ELEMENT OF :LIST:?;
 0 TEST EMPTYP :LIST:
20 IFTRUE OUTPUT "FALSE"
30 IF IS :THING: FIRST :LIST: OUTPUT "TRUE"
40 OUTPUT $MP :THING: BUTFIRST :LIST:
END


TO $ADDERROR :MES: ;UTILITY, ADDS :MES: AS "ERROR" IE TO VALUE OF "(DRIB
                                  **BLE NO) D";
 0 $ADD SENTENCE :DRIBBLE NO: "D" :MES:
END


TO $ADD :PLACE: :MES: ;UTILITY, CONCATENATES :MES: WITH CURRENT VALUE OF
                                  ** :PLACE:;
 0 MAKE :PLACE: SENTENCE THING :PLACE: :MES:
END


TO $ADDSYSC :MES: ;ADDS COMMENT TO CURRENT DRIBBLE FILE LINE (TO VARIABL
                                  **E "(DRIBBLE NO) B");
 0 $ADD SENTENCE :DRIBBLE NO: "B" :MES:
END


TO $RUBOUTP :N: ;UTILITY, WAS LINE FOLLOWING :N: RUBBED OUT? NEEDED FOR
                                  **MULTILINE COMMANDS SUCH AS EDT, MA
                                  **KE ETC;
 0 OUTPUT $MP "RUBOUT" THING SENTENCE :N: "A"
END
```

USER-DEFINABLE PROCEDURES

```
TO $EXAMINE_L ;EMPTY PROCEDURE DEFINABLE BY USER (SEE USERS GUIDE). FILL
                                        **ED IN HERE TO CORRESPOND TO SET OF
                                        ** EXAMPLES.;
10 IF NOT IS FIRST :CURRENT LINE: :CURRENT PROC: STOP
20 IF NOT $MP "RECURSIVE" THING SENTENCE :DRIBBLE NO: "B" $ADD SENTENCE
                                        **:DRIBBLE NO: "B" "RECURSIVE"
END


TO $USEUPEL ;EMPTY,USER DEFINABLE PROCEDURE. SEE USERS GUIDE FOR APPLICA
                                        **TIONS.;
10 OUTPUT "FALSE"
END


TO $EXAMINELINE ;EMPTY,USER DEFINABLE PROCEDURE DESCRIBED IN USERS GUIDE
                                        **;
END


TO $USEUPLINE ;EMPTY, USER DEFINABLE PROCEDURE DESRCIBED IN USERS GUIDE;
 0 OUTPUT "FALSE"
END


TO $ENDLINE ;EMPTY,USER DEFINABLE PROCEDURE DESCRIBED IN USERS GUIDE;
END
```

5. Running

The general procedure structure is again described in the User's Guide. The annotated listings here separate rather neatly into two parts: the graphics part, all subprocedures of $DISPLAY, and the non-graphics, text-oriented commands.

[1]

```
TO $STARTRUN ;INITIALIZES SYSTEM;
5 ERASE ALL NAMES
10 TYPE "DRIBBLE FILE:"
20 DO SENTENCE "GET" REQUEST
30 MAKE "DRIBBLE NO" 0
END
```

[2]

```
TO $SAVEWORLD ;SAVE CURRENT STATUS OF WORK ON DRIBBLE FILE;
10 SAVE THE WORLD
END
```

[3]

```
TO $GETWORLD ;RETRIEVE PREVIOUSLY SAVED WORK STATUS;
10 ERASE ALL NAMES
20 ERASE ALL ABBREVIATIONS
30 PRINT "TYPE: GET THE WORLD"
40 ERASE ALL PROCEDURES
END
```

[4]

```
TO $DOLINE ;DISPLAYS AND EXECUTES ONE DRIBBLE LINE;
5 MAKE "DRIBBLE NO" SUM :DRIBBLE NO: 1
10 TEST $GOODPARSEP :DRIBBLE NO:
11 IF NOT $SHOWLINEP IFTRUE $DISP :DRIBBLE NO:
12 IF NOT $NICEP STOP ELSE IFFALSE STOP
15 MAKE "CURRENT PROCEDURE" FIRST THING SENTENCE :DRIBBLE NO: "C"
20 IF NOT EMPTYP THING SENTENCE :DRIBBLE NO: "C" MAKE SENTENCE SENTENCE
                                    **"GRAPH" :CURRENT PROCEDURE: FIRST
                                    **BUTFIRST THING SENTENCE :DRIBBLE N
                                    **O: "C" BUTFIRST BUTFIRST THING SEN
                                    **TENCE :DRIBBLE NO: "C"
35 DO THING SENTENCE :DRIBBLE NO: "N"
40 IF NOT EITHER $SHOWLINEP EMPTYP :CURRENT PROCEDURE: $GOODLIST :CURREN
                                    **T PROCEDURE: AND - 500 AND - 500
END
```

[5]

```
TO $DOTO :NUM: ;EXECUTES DRIBBLE LINES TO LINE NUMBERED :NUM:;
10 $DOLINE
30 IF GREATERP SUM :DRIBBLE NO: 1 :NUM: STOP
40 IF $EOF STOP
50 $DOTO :NUM:
END
```

[6]

```
TO $DOUNTIL :DESCR: ;EXECUTE DRIBBLE LINES UNTIL :DESCR: IS FOUND IN A C
                                          **OMMENT TAG;
10 $DOLINE
30 IF EITHER $DESCP :DESCR: :DRIBBLE NO: $EOF STOP
50 $DOUNTIL :DESCR:
END
```

[7]

```
TO $DOALL ;EXECUTE ENTIRE DRIBBLE FILE, STARTING AT CURRENT POSITION;
10 $DOLINE
30 IF $EOF STOP
40 $DOALL
END
```

[8]

```
TO $WHERE ;GIVES STATUS OF USER;
10 PRINT SENTENCE "AT DRIBBLE LINE" :DRIBBLE NO:
15 PRINT "#####WHICH IS"
20 PRINT THING SENTENCE :DRIBBLE NO: "N"
30 PRINT SENTENCE SENTENCE "THE PROCEDURES" $UPTOLINE :FINAL CONTENTS: "
                                          **HAVE BEEN DEFINED"
END
```

[9]

```
TO $ALLDESCR ;GIVES LIST OF ALL COMMENTS IN DRIBBLE FILE . IE VALUES OF
                                          **"(N) B";
10 PRINT $ALLDESCR1 1
END
```

[10]

```
TO $ALLDESCR1 :N: ;USED BY $ALLDESCR;
10 IF EMPTYP THING SENTENCE :N: "N" OUTPUT :EMPTY:
20 OUTPUT $UNION ( SENTENCES THING SENTENCE :N: "B" ) ( SENTENCES $ALLDE
                                          **SCR1 SUM :N: 1 )
END
```

[11]

```
TO $STEPPROCEDURE :PROCNAME: ;STEPS THROUGH SUCCESSIVE VERSIONS OF DEFIN
                                    **ITION OF :PROCNAME: ACROSS ENTIRE
                                    **DRIBBLE FILE;
10 $STEPTHROUGH :EMPTY: THING SENTENCE "STRUC" :PROCNAME:
END
```

[12]

```
TO $STEPTHROUGH :LISTONE: :LISTTWO: ;MAIN SUBPROCEDURE OF STEPPROCEDURE;
10 IF EMPTYP :LISTTWO: STOP
20 MAKE "LISTONE" $ADDALINE SENTENCE FIRST :LISTTWO: FIRST BUTFIRST :LIS
                                    **TTWO: :LISTONE:
30 $PRINTLIST :LISTONE: FIRST :LISTTWO:
35 IGNORE REQUEST
40 $STEPTHROUGH :LISTONE: BUTFIRST BUTFIRST :LISTTWO:
END
```

[13]

```
TO $ADDALINE :LINE PAIR: :CURRENT LIST: ;USED BY $STEPPROCEDURE TO PUT N
                                    **EW LINE :LINE PAIR: INTO :CURRENT
                                    **LIST: WHICH IS PREVIOUS STAGE;
  IF EMPTYP :CURRENT LIST: OUTPUT :LINE PAIR:
3 IF IS FIRST BUTFIRST :LINE PAIR: "ERASE" OUTPUT :EMPTY:
6 IF IS FIRST FIRST BUTFIRST :LINE PAIR: "-" OUTPUT $DELETE BUTFIRST FIR
                                    **ST BUTFIRST :LINE PAIR: :CURRENT L
                                    **IST:
10 IF IS FIRST BUTFIRST :LINE PAIR: FIRST BUTFIRST :CURRENT LIST: OUTPUT
                                    **  SENTENCE :LINE PAIR: BUTFIRST BU
                                    **TFIRST :CURRENT LIST:
20 IF GREATERP FIRST BUTFIRST :CURRENT LIST: FIRST BUTFIRST :LINE PAIR:
                                    **OUTPUT SENTENCE :LINE PAIR: :CURRE
                                    **NT LIST:
30 OUTPUT SENTENCE SENTENCE FIRST :CURRENT LIST: FIRST BUTFIRST :CURRENT
                                    ** LIST: $ADDALINE :LINE PAIR: BUTFI
                                    **RST BUTFIRST :CURRENT LIST:
END
```

[14]

```
TO $PRINTLIST :LIST: :CURRENT LINE: ;PRINTS EACH STAGE OF $STEPPROCEDURE
                                    **;
10 IF EMPTYP :LIST: STOP
15 IF IS FIRST :LIST: :CURRENT LINE: TYPE "==>"
20 PRINT THING SENTENCE FIRST :LIST: "N"
30 $PRINTLIST BUTFIRST BUTFIRST :LIST: :CURRENT LINE:
END
```

[15]

```
TO $FINDLINES :FROM: :DESCRIPTOR: ;FINDS ALL OCCURRENCES OF :DESCRIPTOR:
                                    ** IN COMMENT TAGS STARTING AT LINE
                                    **:FROM:;
 0 IF EMPTYP THING SENTENCE :FROM: "N" STOP
20 IF $DESCP :DESCRIPTOR: :FROM: PRINT SENTENCES :DESCRIPTOR: "IN" :FROM
                                    **: "----" THING ( SENTENCE :FROM: "N
                                    **" )
30 $FINDLINES ( :FROM:+1 ) :DESCRIPTOR:
END
```

[16]

```
TO $SUPTOLINE :LIST: ;COMPUTES THAT PART OF :FINAL CONTENTS: WHICH HAS BE
                                    **EN DEFINED AS OF :DRIBBLE NO:;
10 IF EMPTYP :LIST: OUTPUT :EMPTY:
20 IF GREATERP LAST BUTLAST :LIST: :DRIBBLE NO: OUTPUT $UPTOLINE BUTLAST
                                    **  BUTLAST BUTLAST :LIST:
30 IF IS LAST :LIST: "ERASE" OUTPUT $REMOVE LAST BUTLAST BUTLAST :LIST:
                                    **$UPTOLINE BUTLAST BUTLAST BUTLAST
                                    **:LIST:
40 OUTPUT SENTENCE $UPTOLINE BUTLAST BUTLAST BUTLAST :LIST: LAST BUTLAST
                                    **  BUTLAST :LIST:
END
```

[17]

```
TO $GOODLIST :NAME: :X: :Y: ;PRINTS PROCEDURE DEFINITION IN SMALL LETTER
                                    **S ON LOWER LEFT OF DISPLAY. USED T
                                    **O SHOW CURRENT STATUS OF PROCEDURE
                                    ** DEF;
3 WIPE
6 MESSAGE "0 0" ""
10 $GOODLIST1 :NAME: LINES :NAME: :X: :Y:
END
```

[18]

```
TO $GOODLIST1 :NAME: :LL: :X: :Y: ;COMPUTES POSITION OF FIRST LINE FOR $
                                    **GOODLIST;
10 $GOODLIST2 :NAME: :LL: :X: SUM :Y: SUM 12 PRODUCT 12 $COUNTT :LL:
END
```

[19]

```
TO $GOODLIST2 :NAME: :LL: :X: :Y: ;DISPLAYS TITLE LINE FOR $GOODLIST;
 0 MESSAGE SENTENCE :X: SUM :Y: 12 BUTFIRST TEXT :NAME: 0
20 $GOODLIST3 :NAME: :LL: :X: :Y:
END
```

[20]

```
TO $GOODLIST3 :NAME: :LL: :X: :Y: ;PRINTS LINES OF :NAMES: FOR $GOODLIST
                                   **;
10 TEST EMPTYP :LL:
20 IFTRUE MESSAGE SENTENCE :X: :Y: "END"
30 IFTRUE STOP
40 MESSAGE SENTENCE :X: :Y: TEXT :NAME: FIRST :LL:
50 $GOODLIST3 :NAME: BUTFIRST :LL: :X: DIFFERENCE :Y: 12
END
```

GRAPHICS

[21]

```
TO $DISPLAY :ROOT: ;MAIN DISPLAY PROCEDURE;
10 $MAKEALLGRAPHS
20 $NEWROOT :ROOT:
END
```

[22]

```
TO $MAKEALLGRAPHS ;GENERATES A COMPLETE SET OF CURRENT PROCEDURE CONNECT
                                   **IONS;
5 $EMPTYGRAPHS $UPTOLINE :FINAL CONTENTS:
10 $MAKEGRAPH1 $UPTOLINE :FINAL CONTENTS:
END
```

[23]

```
TO $EMPTYGRAPHS :LIST: ;EMPTIES ALL VARIABLES OF FORM "GRAPH (PNAME)" AN
                                   **D "GRAPHT (PNAME)" AS PART OF INIT
                                   **IALIZATION OF DISPLAY;
10 IF EMPTYP :LIST: STOP
15 MAKE SENTENCE "GRAPH" FIRST :LIST: :EMPTY:
20 MAKE SENTENCE "GRAPHT" FIRST :LIST: :EMPTY:
30 $EMPTYGRAPHS BUTFIRST :LIST:
END
```

[24]

```
TO $MAKEGRAPH1 :LIST:
10 IF EMPTYP :LIST: STOP
5 MAKE "LIST" SENTENCE :LIST: :EMPTY:
20 MAKE SENTENCE "GRAPH" FIRST :LIST: $MAKEGRAPH FIRST :LIST: LINES FIRS
                                   **T :LIST:
30 $MAKEGRAPH1 BUTFIRST :LIST:
END
```

[25]

```
TO $MAKEGRAPH :NAME: :LINE LIST: ;USED BY $MAKEGRAPH1 TO CREATE "GRAPH (
                                        **PNAME)" VARIABLE;
10 IF EMPTYP :LINE LIST: OUTPUT :EMPTY:
20 OUTPUT $UNION THING SENTENCE SENTENCE "GRAPH" :NAME: FIRST :LINE LIST
                                        **: $MAKEGRAPH :NAME: BUTFIRST :LINE
                                        ** LIST:
END
```

[26]

```
TO $NEWROOT :ROOT: ;GENERATES NEW DISPLAY STARTING AT :ROOT:;
2 PENUP
5 WIPE
7 MAKE "DISPLAY LIST" :EMPTY:
10 $CTREE :ROOT:
20 $MAKELEVELS :ROOT:
30 $DRAWLEVEL 1 :LEVEL 1: 0
40 $JOINLEVELS :MARKED LIST:
END
```

[27]

```
TO $CTREE :ROOT: ;$CTREE THROUGH $CTREE6 CREATES THAT LIST OF PROCEDURE
                                        **CONNECTIONS ACTUALLY JOINED TO :RO
                                        **OT:;
5 MAKE "MARKED LIST" SENTENCE :ROOT: :EMPTY:
10 $CTREE1 SENTENCES :ROOT:
END
```

[28]

```
TO $CTREE1 :R:
10 TEST EMPTYP :R:
20 IFTRUE STOP
30 $CTREE2 FIRST :R: BUTFIRST :R:
END
```

[29]

```
TO $CTREE2 :R: :S:
20 $CTREE3 :R: ( SENTENCES THING SENTENCE "GRAPH" :R: ) :S:
END
```

[30]

```
TO $CTREE3 :R: :SUC: :S:
10 TEST EMPTYP :SUC:
20 IFTRUE $CTREE1 :S:
30 IFTRUE STOP
40 $CTREE4 :R: FIRST :SUC: BUTFIRST :SUC: :S:
END
```

[31]

```
TO $CTREE4 :R: :FSUC: :RSUC: :S:
10 TEST $CTREE5 :FSUC:
20 IFFALSE MAKE SENTENCE "GRAPHT" :R: SENTENCE THING SENTENCE "GRAPHT" :
                                       **R: :FSUC:
25 IFTRUE $CTREE3 :R: :RSUC: :S:
26 IFFALSE MAKE "MARKED LIST" SENTENCE :MARKED LIST: :FSUC:
30 IFFALSE $CTREE3 :R: :RSUC: SENTENCE :FSUC: :S:
END
```

[32]

```
TO $CTREE5 :X:
10 OUTPUT $CTREE6 :X: :MARKED LIST:
END
```

[33]

```
TO $CTREE6 :X: :L:
10 TEST EMPTYP :L:
20 IFTRUE OUTPUT "FALSE"
30 TEST IS :X: FIRST :L:
40 IFTRUE OUTPUT "TRUE"
50 OUTPUT $CTREE6 :X: BUTFIRST :L:
END
```

[34]

```
TO $MAKELEVELS :ROOTPROC: ;CREATES THE LEVEL LISTS FOR DISPLAY, EACH LEV
                                       **EL BEING A LIST OF PNAMES WITH COO
                                       **RDINATES :ROOT: IS LEVEL 1;
10 $ZEROLEVEL 1
15 MAKE "LEVEL 1" SENTENCE SENTENCE - 42 :ROOTPROC: 42
20 $MAKELEVEL 2 0 THING SENTENCE "GRAPHT" :ROOTPROC:
END
```

[35]

```
TO $ZEROLEVEL :N: ;EMPTIES ALL VARIABLES OF FORM "LEVEL (N)";
10 IF EMPTYP THING SENTENCE "LEVEL" :N: STOP
20 MAKE SENTENCE "LEVEL" :N: :EMPTY:
30 $ZEROLEVEL ( :N:+1 )
END
```

[36]

```
TO $MAKELEVEL :LEVEL NUM: :PIVOT: :LIST: ;CREATES LEVEL LISTS FOR $MAKEL
                                     **EVELS;
10 IF EMPTYP :LIST: STOP
20 MAKE "LIST" SENTENCE ( $NEARESTHOLE 0 THING SENTENCE "LEVEL" :LEVEL N
                                     **UM: :PIVOT: ) :LIST:
30 MAKE SENTENCE "LEVEL" :LEVEL NUM: SENTENCE SENTENCE SENTENCE THING SE
                                     **NTENCE "LEVEL" :LEVEL NUM: ( FIRST
                                     ** :LIST: ) - 42 $NOTE FIRST BUTFIRS
                                     **T :LIST: ( FIRST :LIST: ) + 42
40 $MAKELEVEL ( :LEVEL NUM:+1 ) FIRST :LIST: THING SENTENCE "GRAPHT" FIR
                                     **ST BUTFIRST :LIST:
50 $MAKELEVEL :LEVEL NUM: :PIVOT: BUTFIRST BUTFIRST :LIST:
END
```

[37]

```
TO $NEARESTHOLE :X: :LIST: :PIVOT: ;USED BY $MAKELEVEL TO FIND GAPS BETW
                                     **EEN BOXES ALREADY DEFINED, ON :LIS
                                     **T:. PIVOT IS CENTER;
10 IF EMPTYP :LIST: OUTPUT ( :PIVOT:-20 )
20 TEST NOT $INSIDEP ( :PIVOT:+:X:-42 ) ( :PIVOT:+:X:+42 ) :LIST:
30 IFTRUE OUTPUT :PIVOT:+:X:
40 TEST NOT $INSIDEP ( :PIVOT:-:X:-42 ) ( :PIVOT:-:X:+42 ) :LIST:
50 IFTRUE OUTPUT :PIVOT:-:X:
60 OUTPUT $NEARESTHOLE :X:+7 :LIST: :PIVOT:
END
```

[38]

```
TO $INSIDEP :A: :B: :LIST: ;USED BY $NEARESTHOLE TO CHECK OVERLAP;
10 IF EMPTYP :LIST: OUTPUT "FALSE"
20 IF EITHER $BETWEENP :A: :B: FIRST :LIST: $BETWEENP :A: :B: FIRST BUTF
                                     **IRST BUTFIRST :LIST: OUTPUT "TRUE"
30 OUTPUT $INSIDEP :A: :B: BUTFIRST BUTFIRST BUTFIRST :LIST:
END
```

[39]

```
TO $BETWEENP :A: :B: :TEST: ;IS :TEST: BETWEEN :A: AND :B:?;
5 IF EITHER IS :A: :TEST: IS :B: :TEST: OUTPUT "TRUE"
10 OUTPUT EITHER BOTH GREATERP :A: :TEST: GREATERP :TEST: :B: BOTH GREAT
                                     **ERP :B: :TEST: GREATERP :TEST: :A:
END
```

```
  [40]

TO $DRAWLEVEL :N: :LIST: :VERT: ;DOES POSITIONAL COMPUTATION AND ACTUAL
                                          **DRAWING OF BOXES ON EACH LEVEL OF
                                          **DISPLAY;
10 IF EMPTYP THING SENTENCE "LEVEL" :N: STOP
20 TEST EMPTYP :LIST:
30 IFTRUE $DRAWLEVEL :N:+1 THING SENTENCE "LEVEL" ( :N:+1 ) :VERT: - ( $
                                          **MAXHEIGHT THING SENTENCE "LEVEL" :
                                          **N: ) - 40
40 IFTRUE STOP
50 $BOXIN FIRST BUTFIRST :LIST: ( FIRST :LIST: ) + 12 :VERT:
55 MAKE "DISPLAY LIST" SENTENCES :DISPLAY LIST: ( FIRST BUTFIRST :LIST:
                                          **) :N: ( FIRST :LIST: ) + 42 :VERT:
                                          ** ( FIRST :LIST: ) + 42 :VERT: - $H
                                          **EIGHT FIRST BUTFIRST :LIST:
60 $DRAWLEVEL :N: BUTFIRST BUTFIRST BUTFIRST :LIST: :VERT:
END

  [41]

TO $MAXHEIGHT :LIST: ;COMPUTES MAXIMUM HEIGHT OF THE BOXES GIVEN BY :LIS
                                          **T:;
5 IF EMPTYP :LIST: OUTPUT 0
10 OUTPUT MAXIMUM ( $HEIGHT FIRST BUTFIRST :LIST: ) $MAXHEIGHT BUTFIRST
                                          **BUTFIRST BUTFIRST :LIST:
END

  [42]

TO $HEIGHT :STRING: ;COMPUTES HEIGHT OF TEXT GIVEN 8 CHARACTER WIDTH;
10 OUTPUT  2 * ( ( ( COUNT :STRING: ) + 7 ) / 8 ) = 5
END

  [43]

TO $BOXIN :NAME: :UPPERLHX: :UPPERLHY: ;BOXES IN :NAME: GIVEN COORDS OF
                                          **UPPER LEFT HAND CORNER;
10 MOVE SENTENCE :UPPERLHX:-3 :UPPERLHY:+3
20 PENDOWN
30 MOVE SENTENCE :UPPERLHX:+63 :UPPERLHY:+3
40 MOVE SENTENCE :UPPERLHX:+63 :UPPERLHY: - ( $HEIGHT :NAME: ) - 9
50 MOVE SENTENCE :UPPERLHX:-3 :UPPERLHY: - ( $HEIGHT :NAME: ) - 9
60 MOVE SENTENCE :UPPERLHX:-3 :UPPERLHY:+3
70 $ENCLOSEMES :NAME: :UPPERLHX:+4 :UPPERLHY:-9
80 PENUP
END
```

[44]

```
TO $ENCLOSEMES :NAME: :X: :Y: ;PUTS MESSAGE IN BOX;
 0 TEST GREATERP 8 COUNT :NAME:
 3 IFTRUE $CENTER :NAME: :X: :Y:
 6 IFTRUE STOP
20 MESSAGE SENTENCE :X: :Y: $PULL 8 :NAME:
30 $ENCLOSEMES $DELETE. 8 :NAME: :X: :Y:-12
END
```

[45]

```
TO $CENTER :NAME: :X: :Y: ;USED BY $ENCLOSEMES TO CENTER TEXT;
10 IF EMPTYP :NAME: STOP
20 MESSAGE SENTENCE SUM :X: ( 7 * ( 8 - COUNT :NAME: ) ) / 2 :Y: :NAME:
END
```

[46]

```
TO $JOINLEVELS :LIST: ;JOINS DISPLAYED BOXES. ITERATES THROUGH LEVELS;
10 IF EMPTYP :LIST: STOP
20 $JOINLINES FIRST :LIST: THING SENTENCE "GRAPH" FIRST :LIST:
30 $JOINLEVELS BUTFIRST :LIST:
END
```

[47]

```
TO $JOINLINES :NAM: :LIST: ;DRAWS CONNECTIONS BETWEEN BOXES, ITERATES TH
                                                **ROUGH ELEMENTS;
10 IF EMPTYP :LIST: STOP
20 $JOIN :NAM: FIRST :LIST:
30 $JOINLINES :NAM: BUTFIRST :LIST:
END
```

[48]

```
TO $JOIN :A: :B: ;CONNECTS BOX :A: AND BOX :B: WITH ARC OR ARROW;
1 IF IS :A: :B: STOP
3 MAKE "A" $COORD :A: :DISPLAY LIST:
6 MAKE "B" $COORD :B: :DISPLAY LIST:
10 TEST IS FIRST :A: FIRST :B:
20 IFTRUE $ARC BUTFIRST :A: BUTFIRST :B:
30 IFTRUE STOP
40 $JOINLINE :A: :B:
END
```

[49]

```
TO $JOINLINE :A: :B: ;JOINS BOX :A: AND BOX :B: WITH ARROW;
10 TEST LESSP FIRST :A: FIRST :B:
20 IFTRUE $DRAWLINE BUTFIRST BUTFIRST BUTFIRST :A: SENTENCE FIRST BUTFIR
                                             **ST :B: FIRST BUTFIRST BUTFIRST :B:
30 IFTRUE STOP
40 $DRAWLINE SENTENCE FIRST BUTFIRST :A: FIRST BUTFIRST BUTFIRST :A: BUT
                                             **FIRST BUTFIRST BUTFIRST :B:
END
```

[50]

```
TO $ERASEALL ;PSEUDO ERASE ALL TO PRESERVE SYSTEM;
10 $ERASEALL1 SENTENCES $UPTOLINE :FINAL CONTENTS:
END
```

[51]

```
TO $ERASEALL1 :PRS: ;USED BY $ERASEALL;
5 IF EMPTYP :PRS: STOP
10 DO SENTENCE "ERASE" FIRST :PRS:
20 $ERASEALL1 BUTFIRST :PRS:
END
```

UTILITY PROCEDURES

```
TO $YESP :ANS: ;;
10 IF IS :ANS: "Y" OUTPUT "TRUE"
20 IF IS :ANS: "YES" OUTPUT "TRUE"
30 OUTPUT "FALSE"
END

TO $SKIP :N:
5 IF ZEROP :N: STOP
10 PRINT ""
20 $SKIP DIFFERENCE :N: 1
END

TO $DISP :NO: ;PRINTS DRIBBLE LINE :NO:;
10 PRINT SENTENCES "***" :NO: "***" THING SENTENCE :NO: "N"
END

TO $ADD :PLACE: :MES: ;CONCATENATES VALUE OF :PLACE: AND :MES:;
10 MAKE :PLACE: SENTENCE THING :PLACE: :MES:
END
```

```
TO $COUNTT :EL: ;COUNT WITH KLUDGY FIX OF BUG IN COUNT 0;
 0 IF EMPTYP :EL: OUTPUT 0 ELSE OUTPUT COUNT :EL:
END


TO $PULL :N: :LIST: ;FIRST :N: OF :LIST:;
10 IF ZEROP :N: OUTPUT :EMPTY:
20 OUTPUT WORD FIRST :LIST: $PULL :N:-1 BUTFIRST :LIST:
END

TO $DELETE. :N: :LIST: ;BUTFIRST :N: OF :LIST:;
10 IF :N:=0 OUTPUT :LIST:
20 OUTPUT $DELETE. :N:-1 BUTFIRST :LIST:
END


TO $MP :EL: :LIST: ;MEMBERP;
5 MAKE "LIST" SENTENCES :LIST:
10 IF EMPTYP :LIST: OUTPUT "FALSE"
20 IF IS :EL: FIRST :LIST: OUTPUT "TRUE"
30 OUTPUT $MP :EL: BUTFIRST :LIST:
END

TO $MINUS :X:
 0 IF IS FIRST :A: "-" OUTPUT BUTFIRST :X:
20 OUTPUT WORD "-" :X:
END


TO $COORD :PNAME: :LIST: ;FIND COORDINATES OF :PNAMES: ON LIST OF QUINTU
                                         **PLES;
20 IF IS $NOTE :PNAME: FIRST :LIST: OUTPUT $F5 BUTFIRST :LIST:
30 OUTPUT $COORD :PNAME: BUTFIRST BUTFIRST BUTFIRST BUTFIRST BUTFIRST BU
                                         **TFIRST :LIST:
END


TO $F5 :N:
10 OUTPUT SENTENCE SENTENCE SENTENCE SENTENCE FIRST :N: FIRST BUTFIRST :
                                         **N: FIRST BUTFIRST BUTFIRST :N: FIR
                                         **ST BUTFIRST BUTFIRST BUTFIRST :N:
                                         **FIRST BUTFIRST BUTFIRST BUTFIRST B
                                         **UTFIRST :N:
END
```

```
TO $ARC :A: :B: ;DRAWS ARC FROM POINT :A: TO HORIZONTAL POINT :B:;
10 $SEMICIRCLE SENTENCE FIRST :A: FIRST BUTFIRST :A: QUOTIENT ( ( FIRST
                                         **:B: ) - FIRST :A: ) 2

20 PENDOWN
30 LEFT 10
40 BACK 15
50 FRONT 15
60 RIGHT 20
70 BACK 15
80 PENUP
END

TO $SEMICIRCLE :PT: :RADIUS:
10 MOVE :PT:
15 SETHEADING 90
20 PENDOWN
30 $REPEAT 13 SENTENCE "FRONT" 10 * $ABS :RADIUS:/38 SENTENCE "RIGHT" (
                                         **$SGN :RADIUS: ) * 15

40 PENUP
END

TO $REPEAT :N: :A: :B:
10 IF ZEROP :N: STOP
20 DO :A:
30 DO :B:
40 $REPEAT ( :N:-1 ) :A: :B:
END

TO $ABS :X:
10 TEST IS FIRST :X: "-"
20 IFTRUE OUTPUT BUTFIRST :X:
30 OUTPUT :X:
END

TO $SGN :A:
10 IF IS FIRST :A: "-" OUTPUT - 1
20 OUTPUT
END

TO $UNION :LISTONE: :LISTTWO:
3 MAKE "LISTONE" SENTENCES :LISTONE:
6 MAKE "LISTTWO" SENTENCES :LISTTWO:
10 IF EMPTYP :LISTONE: OUTPUT :LISTTWO:
20 IF $MP FIRST :LISTONE: :LISTTWO: OUTPUT $UNION BUTFIRST :LISTONE: :LI
                                         **STTWO:
30 OUTPUT $UNION BUTFIRST :LISTONE: SENTENCE FIRST :LISTONE: :LISTTWO:
END
```

```
TO $DRAWLINE :A: :B: ;DRAWS ARROW FROM POINT :A: TO POINT :B:;
 0 MOVE :A:
20 PENDOWN
30 MOVE :B:
40 LEFT 10
50 BACK 15
60 FRONT 15
70 RIGHT 20
80 BACK 15
90 PENUP
END


TO $REMOVE :EL: :LIST: ;STRIPS FRONT OF :LIST: UP TO AND INCLUDING :EL:;
10 MAKE "LIST" SENTENCES :LIST:
20 IF IS :EL: FIRST :LIST: OUTPUT BUTFIRST :LIST:
30 OUTPUT $REMOVE :EL: BUTFIRST :LIST:
END


TO $DELETE :EL: :LIST: ;PAIRWISE DELETE;
20 IF IS :EL: FIRST BUTFIRST :LIST: OUTPUT BUTFIRST BUTFIRST :LIST:
30 OUTPUT SENTENCES FIRST :LIST: FIRST BUTFIRST :LIST: $DELETE :EL: BUTF
                                         **IRST BUTFIRST :LIST:

END


TO $DESCP :DESCR: :LINE NO: ;IS DESCRIPTOR :DESCR: IN COMMENT OF :LINE N
                                         **O:? (IE IN "(LINE NO) B");
 0 OUTPUT $MP :DESCR: THING SENTENCE :LINE NO: "B"
END


TO $GOODPARSEP :NUM: ;HAS DRIBBLE LINE :NUM: PARSED CORRECTLY?;
 0 OUTPUT EMPTYP THING SENTENCE :NUM: "D"
END


TO $EOF ;AT END OF DRIBBLE FILE?;
 0 TEST EMPTYP THING SENTENCE :DRIBBLE NO: "N"
20 IFTRUE PRINT "****END-OF-FILE****"
30 IFTRUE OUTPUT "TRUE"
40 OUTPUT "FALSE"
END
```

USER-DEFINABLE PROCEDURES

```
TO $LOOKAHEAD ;DEFINED IN EXAMPLE TO PERMIT LOOKAHEAD IN DRIBBLE FILE AN
                                           **D MARKING OF IGNORABLE LINES;
 0 $LOOKAHEAD1 1
END

TO $LOOKAHEAD1 :N: ;USED BY $LOOKAHEAD.;
 0 TYPE "MORE?..."
20 TEST $YESP REQUEST
30 IFFALSE PRINT SENTENCES "**RESUME AT***" :DRIBBLE NO: "***"
40 IFFALSE $SKIP 2
50 IFFALSE DO THING SENTENCE :DRIBBLE NO: "N"
60 IFFALSE STOP
70 TYPE "[[[[["
80 $DISP SUM :DRIBBLE NO: "N"
90 TYPE "IGNORE?..."
100 IF $YESP REQUEST $ADD SENTENCE SUM :DRIBBLE NO: :N: "D" "IGNORE"
110 $LOOKAHEAD1 SUM :N: 1
END


TO $NOTE :PROC: ;EMPTY,USER DEFINABLE PROCEDURE, HERE FILLED IN AS IN EX
                                           **AMPLE;
10 IF $RECURSEP :PROC: OUTPUT WORD WORD "**" :PROC: "**"
20 OUTPUT :PROC:
END

TO $RECURSEP :PROC: ;USED DEFINED IN EXAMPLE TO CHECK RECURSIVENESS;
10 OUTPUT $MP :PROC: :RECURSIVE LIST:
END


TO $SHOWLINEP
 0 OUTPUT "FALSE"
END


TO $NICEP ;EMPTY, USER DEFINABLE PROCEDURE, HERE FILLED IN AS IN EXAMPLE
                                           **;
 0 IF $MP "RECURSIVE" THING SENTENCE :DRIBBLE NO: "B" MAKE "RECURSIVE LI
                                           **ST" $UNION :CURRENT PROCEDURE: :RE
                                           **CURSIVE LIST:
20 OUTPUT "TRUE"
END
```

6. The Display Facility

To facilitate the display of program structure diagrams, a display system was implemented on the IMLAC PDS-1 using the IMLAC executive program. The apparently rather idiosyncratic nomenclature, SETTURTLE for example, arises from the fact that we chose our primitives to be a superset of the commands controlling our robot "turtle". (These are described in the Appendix.)

The display screen is considered to be 1024 by 1024 units, with the origin in the center of the screen. The "turtle" itself is in the shape of an isosceles triangle whose base is 8 and altitude is 16. The sharp end of the triangle points toward the heading of the turtle. Headings are in degrees, from the horizontal. The position of the turtle is kept to more accuracy internally than is recorded on the screen. This procedure avoids undesirable round-off errors.

The turtle also has a "pen" which is initially "up". If the pen is down, any command that changes the position of the turtle (except for HOME and the SET commands) will draw a line from the initial position to the final position.

The following commands change position:
FRONT, BACK, MOVE, SETXY, SETX, SETY, HOME, SETTURTLE.

The following commands change heading:
RIGHT, LEFT, SETHEADING, SETTURTLE.

The following function gives information about the turtle status:
HERE.

6.1 Description of Commands and Functions

FRONT takes one argument: a numerical string. It moves the turtle forward by the number of units indicated, in the direction the "turtle" is pointed. If the "pen" is "down", it draws a line between the initial and final positions.

BACK :X: is the equivalent of FRONT (-:X:).

LEFT takes one argument: a numerical string. It changes the heading angle of the turtle by adding its argument to the current heading and reducing modulo 360. The orientation of the "turtle" (i.e., the vertex of the triangle) is changed to the new heading.

RIGHT :X: is equivalent to LEFT (-:X:).

SETHEADING takes one argument: a numerical string. It changes the heading of the turtle (as in LEFT) to the argument reduced modulo 360, and changes the orientation appropriately.

SETX takes one argument: a numerical string. It changes the X-coordinate of the turtle to the argument. The orientation of the turtle is not changed, and no vector is drawn.

SETY behaves like SETX, except that the Y-coordinate is involved.

SETXY takes one argument: a sentence having two numerical words. It changes the X- and Y-coordinates respectively to the first and second words, as in SETX.

SETTURTLE takes one argument: a sentence having two three-digit words. It changes the (X,Y) coordinates of the turtle as in SETXY, using the first two words, and the heading as in SETHEADING using the third word.

HOME is the same as SETTURTLE "0 0 0".

WIPE erases all lines drawn on the display and all messages (see below) but leaves the turtle in the same position and orientation as it was before the command was executed.

PENUP "raises the pen". It causes no command to draw a vector until a PENDOWN command is executed.

PENDOWN "lowers the pen". It causes the commands FRONT, BACK, and MOVE to draw vectors from the initial to final positions of the turtle, until a PENUP command is executed.

MOVE takes one argument in identical format and meaning as SETXY. It causes the same action as SETXY. In addition, it changes the orientation of the turtle to point in the direction of motion, and draws a line if the "pen is down".

MESSAGE takes two inputs: the first is a sentence as in SETXY, indicating a position; the second is a sentence or word which is interpreted as a string. The characters in the second argument are displayed horizontally.

HERE has no arguments. It is a function which has a value equal to a sentence of three words. The first two words represent the (X, Y) coordinates of the turtle, and the third word represents the heading of the turtle. In other words, the output of HERE is in the same format as the input to SETTURTLE.

6.2 Specific Implementation on the IMLAC

The IMLAC contains a central processor (similar to a DEC PDP-9) and a display processor (with long vector drawing) as well as 8K (of 16-bit words) of memory. The display processor periodically (60 times per second) refreshes the display by executing a sequence of vector drawing commands.

The TENEX-IMLAC implementation operates by directly modifying the display program inside the IMLAC.

For other types of display processors, such as storage tubes, or the PLATO terminal or refresh scopes without a central processor, a different strategy must be used. For the first two alternatives, display lists need not be kept. As vectors are generated, the properly formatted display instructions are merely transmitted to the scope. For a storage tube without selective erase, such as a COMPUTEK, a change should probably be made to the "turtle indicator", i.e., the small triangle indicating position and bearing of the turtle. If there is a turtle indicator, there would be "tracks" left on the display, that is, images of old turtle indicators. The procedure in this case would be either to eliminate the turtle indication altogether or to use a programable cursor to indicate position only. If the storage tube has selective erase, then a turtle indicator can still be drawn, but must be "remembered" in order to erase previous indications.

A refresh tube without memory can be handled in much the same way as the IMLAC, except that the display lists should be kept inside the main computer.

There are some display processors, such as other models of the IMLAC, which do not have "long vector" drawing hardware. More precisely, this means that vectors of arbitrary length cannot be drawn with one display processor instruction. When using a processor lacking this capability, additional programs must be written. These programs will convert a vector specification into a series of instructions for the display processors. Usually the basic display instruction will be able to draw "short vectors" - vectors whose length is less than 3 or 4 units, where the entire screen is 1024 units wide.

The IMLAC driving programs used in this project were written by Victor S. Miller in MACRO, the assembly language of the PDP-10. The IMLAC programs themselves were also written on the PDP-10 using an in-house IMLAC assembler.

Part 3.

Analysis Package

1. Introduction to Analysis Package

The analysis facilities described in the user's guide are very general and are not customized. We describe here the construction of extended facilities for use in various aspects of the analysis -- working with student programs directly, augmenting the system's parsing capabilities, and extending the system's run-time capabilities. These facilities constitute our analysis package.

At the outset a teacher or researcher will find the capabilities of the dribble file analysis system very substantial. The initial command structure and associated semantics will probably seem reasonable and adequate. Continuing use of the system, particularly when the use is intense, will likely lead to some dissatisfaction, both with the command structure and its interpretation. The serious user will want to personalize and extend the specific information that the parser generates. He will want to incorporate his own ideas on editing, error correction, and execution facilities. It is precisely for this reason that the system was written in LOGO, a relatively simple and accessible, yet powerful and easily extensible language. In this section we will discuss possible user extensions of various kinds.

2. User Definition of Analysis Procedures

We first discuss those "advanced" features of LOGO which, although originally added for work with sophisticated students, are very valuable in extending the dribble file system. There is no distinction in LOGO as there is in some programming languages between system and nonsystem commands. Thus, as a trivial example, we can erase a set of LOGO procedures in two different ways. We can use the LOGO ERASE command directly or we can write our own procedure for erasing a list of procedures given as input:

```
TO ERASEMANY :LIST:
10 IF EMPTYP :LIST: THEN STOP
20 DO SENTENCE "ERASE" FIRST OF :LIST:
30 ERASEMANY BUTFIRST OF :LIST:
END
```

Also, instead of defining a new LOGO procedure in the usual way, we can define a procedure which creates a new procedure:

```
TO CREATE
10 DO "TO FOO"
20 DO "10 PRINT RANDOM"
30 DO "END"
END
```

CREATE defines the procedure FOO which simply has the effect of printing a random digit. The process is carried out as follows.

```
←CREATE
FOO DEFINED
←LIST FOO

TO FOO
10 PRINT RANDOM
END

←FOO
7
←FOO
2
←
```

The single-input DO command evaluates its input, i.e., executes its input as a LOGO instruction line. The use of DO is, of course, essential in examples like the above where we modify existing LOGO procedures or define new ones. In order to effectively modify procedures, however, we also need to have program access to their current state. The LOGO built-in operations LINES and TEXT make this possible. LINES takes one input, which must be the name of a procedure in the user's workspace, and outputs a sentence composed of the line numbers of that procedure. Using LINES with the procedure CREATE, defined above, for example, we get:

```
←PRINT LINES "CREATE"
0 10 20 30
←
```

(Note the line number 0 which represents the title line.) Given a procedure name and a line number, one can get the entire content of the line (including the line number) using the two-input LOGO operation TEXT. Thus:

```
←PRINT TEXT "CREATE" 10
10 DO "TO FOO"
←PRINT TEXT "CREATE" 0
TO CREATE
←
```

As we will see in the following pages, DO, LINES, and TEXT, combined with the other LOGO primitives, give a user considerable power for extension of the dribble file analysis system. We begin with some simple procedures to augment the basic parsing capabilities built into the analysis system.

3. User Augmentation of the Parsing Procedures

The built-in parsing procedures interpret the student's LOGO program in very much the way that LOGO itself does. From the standpoint of the analyst, however, this process can be improved in various ways so as to run more smoothly or to give him additional information for later phases of the analysis. Examples of each kind are developed next.

3.1 Putting parenthesis-checking into the parsing procedure

In the current implementation of LOGO, balancing of parentheses is not checked as the expression is interpreted. The execution of an unbalanced expression generates an error comment and halts the system. Such halts can be annoying when the user of the analysis system is not interested at the level of detail of local syntax errors. A suitable comment entered in the dribble file during the parsing phase can be used to inhibit execution of such lines as the user sweeps through the dribble file subsequently. Consistent with our general conventions of usage, such a comment is entered into the global variable "(dribble file line no.) B". This is done using the existing one-input procedure $ADDSYSC :MESSAGE:, together with filling in the empty procedures designed to make such additions easy. Thus, we use the global "PAR COUNT" to keep track of depth (its name is a sentence as we require by convention for all dribble file analysis globals). "CURRENT LINE" contains what is left of the line being parsed. We initialize -

```
TO $EXAMINELINE
10 MAKE "PAR COUNT" 0
END
```

We look at each element to see if it is a right or left parenthesis and, if so, take suitable action:

```
TO $EXAMINEEL
10 TEST IS FIRST :CURRENT LINE: "("
20 IFTRUE MAKE "PAR COUNT" SUM OF
     :PAR COUNT: AND 1
30 IFTRUE STOP
40 TEST IS FIRST :CURRENT LINE: ")"
50 IFTRUE MAKE "PAR COUNT" DIFF OF :PAR COUNT: AND 1
60 IF GREATERP 0 :PAR COUNT:
     $ADDSYSC "MATCHING-PARENS"
END
```

And, finally, to terminate the line being parsed we fill in $ENDLINE.

```
TO $ENDLINE
10 IF NOT ZEROP :PAR COUNT:
     $ADDSYSC WORDS
     "MISSING-" :PAR COUNT: "-PARENS"
END
```

3.2 Checking for operation vs. command

It is very useful for later analysis, and in fact very easy during the parsing phase, to generate the specification of whether a procedure is an operation or a command -- whether it merely stops or hands back information to the procedure which called it. We can look for a STOP or an OUTPUT in the procedure definition. If neither exists, the procedure terminates on the END command and is a command. This, of course, is not a perfect algorithm, even within the limits imposed by the halting problem. The simplest form of (syntactic) ambiguity is between OUTPUTting and falling through to the END. Also, considerable benefit could be derived by tracing out GOTOLINE statements (except when their arguments are generated at runtime). But, the simplest approach yields generally satisfactory results for the parsing phase of analysis. More complex analyses are best left for the running phase of dribble file analysis. So, given the type of information we want to find, where should we put it once we find it? Remembering that the form of a procedure is time-dependent, it seems necessary to maintain a running record of the state of the procedure. We define a new data type "(pname) FORM" to contain such information for each procedure defined in the dribble file. At the end of parsing, it might look like

"FOO FORM" IS "137 10 OUTPUT 138 20 STOP 139 10 ERASE"

The value of "FOO FORM" is a set of triples - the dribble file line number, procedure line number, and the relevant command contained. This will enable the use of a new procedure in the running phase to determine the state of any procedure at any point in the dribble file. It is easy to incorporate in the parser procedures which generate these names and values. We need simply define versions of $EXAMINEEL, $EXAMINELINE, and $ENDLINE for this purpose as follows -

```
TO $EXAMINEEL
10 IF EMPTYP :CURRENT PROC: STOP
20 IF EMPTYP :LINE NO: STOP
30 IF NOT EITHER IS FIRST :CURRENT LINE: "OUTPUT"
                 IS FIRST :CURRENT LINE: "STOP"
   STOP
40 $ADD "CURRENT FORM" SENTENCES
        :DRIBBLE NO:
        :LINE NO:
        FIRST :CURRENT LINE:
END
```

where we are using :LINE NO: to keep the line number (if any) of the procedure line currently being defined in the dribble file.

To do this we fill in the definition of $EXAMINELINE.

```
TO $EXAMINELINE
10 IF NUMBERP FIRST :CURRENT LINE:
    MAKE "LINE NO" FIRST :CURRENT LINE:
END
```

The reason, of course, that we have been maintaining the additions to the form statement of FOO separate in "CURRENT FORM" is that we must wait to see if the line parses correctly. If it does not, this new information is simply discarded. We note that $GOODPARSEP is available to do this.

```
TO $ENDLINE
10 IF BOTH $GOODPARSEP :DRIBBLE NO:
    NOT EMPTYP :CURRENT PROC:
    $ADD SENTENCE :CURRENT PROC:
        "FORM"
        :CURRENT FORM:
20 MAKE "CURRENT FORM" :EMPTY:
END
```

As a further extension of this special checking facility, we must include the effects of student erase commands contained in the dribble file. If a line or the whole procedure are erased, this occurrence must be indicated by incorporating a comment so signifying. It is easy to incorporate a check for such erasures which includes appropriate additions to the variables "FOO FORM". This can be done either from scratch or by using results from $PARSEERASE, the top-level parsing procedure concerned explicitly with the ERASE command. These procedures are as straightforward as the ones just developed for checking for operation vs. command. Their implementation is left to the reader.

4. Aids for Execution and Debugging of Student Procedures

In addition to the parsing phase procedures, the analysis package includes aids for running and testing the student's procedures. These are described in the following sections.

4.1 Adding Breakpoints

Insertion of breakpoints into defective, or possibly defective, procedures is a time honored debugging device. The BREAK-GO-CANCEL commands in LOGO give a limited amount of breakpoint control, but, as we will see, they can easily be extended by user-defined procedures to provide fairly general and powerful debugging aids. These insertions can be done "by hand" via insertion of suitable code, or automatically via system calls. The former is good for occasional use; if the dribble file user inserts breakpoints frequently, he may want to write assisting procedures. In particular, cataloging of breakpoints is useful as well as checks to make sure the breakpoint insertion is not destroying anything. A simple way to put in a breakpoint is by just putting in the procedure - BP (pname) $ (line no).* The list of extant breakpoints can be kept in "BREAK POINTS", say, as the pairs (pname) (line no). So, we can write the simple elicitation dialogue

```
TO $INSBREAK
10 $INSBREAK1 $ACCEPTLINE $ACCEPTPNAME
END

TO $ACCEPTPNAME
10 TYPE "INTO PNAME..."               (finds out pname)
20 OUTPUT $ACCEPTPNAME1 REQUEST
END
```

---

[*] The name BP (pname) $ (line no) is chosen to ensure uniqueness.

```
TO $ACCEPTPNAME1 :PNAME:           (validates :PNAME:)
10 IF EMPTYP :PNAME: EXIT "EMPTYPNAME,
     LEAVING $INSBREAK"
20 IF $MP :PNAME: :CONTENTS: OUTPUT :PNAME:
30 PRINT "NO SUCH PROCEDURE"
40 OUTPUT $ACCEPTPNAME1 REQUEST
END


TO $ACCEPTLINE :PNAME:               (elicits line number)
10 TYPE "LINE NO..."
20 OUTPUT SENTENCE :PNAME: AND $ACCEPTLINE1 REQUEST
END

TO $ACCEPTLINE1 :LINE NO:            (checks on line number)
10 IF EMPTYP :LINE NO: EXIT "EMPTY LINE NO, LEAVING
      $INSBREAK"
20 TEST LESSP :LINE NO: 1
30 IFTRUE PRINT "LINE NO MUST BE GREATER THAN 0"
40 IFTRUE OUTPUT $ACCEPTLINE1 REQUEST
50 IF NOT $MP :LINE NO: LINES :PNAME: OUTPUT :LINE NO:
60 PRINT "ALREADY OCCUPIED, WANT TO CLOBBER IT?"
70 IF $YESP REQUEST OUTPUT :LINE NO:
80 DO SENTENCE "LIST" :PNAME:
90 OUTPUT $ACCEPTLINE REQUEST
END


TO $YESP :L:
10 IF $MP :L: "YES Y OK" OUTPUT "TRUE"
20 IF $MP :L: "NO N NAH" OUTPUT "FALSE"
30 PRINT "YES OR NO?"
40 OUTPUT $YESP REQUEST
END
```

The program now has a procedure name and line number, which have been pretty carefully checked out, and is ready to start work via $INSBREAK1. We assume there are just two kinds of breaks, one after a specified number of times through the breakpoint, the other a conditional expression evaluating to "TRUE".

```
TO $INSBREAK1 :PROC LINE:               (inserts breakpoint)
10 DO SENTENCE "EDIT" FIRST             (opens procedure for
     ;PROC LINE:                         insertion of breakpoint)
20 $INSBREAK2 LAST :PROC LINE:
     WORDS "$BP$" FIRST :PROC LINE: "$"
     LAST :PROC LINE:
END
```

```
TO $INSBREAK2 :LINE: :PNAME:
10 DO SENTENCE :LINE: :PNAME:           (putting in the
20 DO "END"                              breakpoint)
30 DO SENTENCE "TO" :PNAME:
40 PRINT "(COUN)TER OR (COND)ITIONAL BREAKPOINT?"
50 IF IS REQUEST "COUN" $MAKECOUNTER ELSE
         $MAKECONDITIONAL
60 DO "END"
70 $ENTERBREAK
END
```

We make a simple counter rather than one which increments on a condition.

```
TO $MAKECOUNTER
10 DO SENTENCES "10 MAKE"               (We are using a unique
     :QUOTE: :PNAME: :QUOTE:             construction for variable
     "$SUMM 1 AND :" :PNAME: ":"         name as well as procedure
                                         name)
20 PRINT "HOW MANY TIMES THROUGH?"
30 DO SENTENCES
     "20 TEST IS :" :PNAME: ":"
     $ACCEPTNUM REQUEST
40 DO SENTENCES
     "30 IFTRUE MAKE"
     :QUOTE: :PNAME: :QUOTE:
     ":EMPTY:"                          (reset counter)
50 DO SENTENCES
     "40 IFTRUE PRINT" :QUOTE:
     "BREAK AT LINE" LAST :PROC LINE:
     "OF" FIRST :PROC LINE: :QUOTE:
60 DO
     "50 IFTRUE BREAK"
70 DO "END"
END
```

```
TO $ACCEPTNUM :NUM:                  (elicits a number)
10 IF NUMBERP :NUM: OUTPUT :NUM:
20 PRINT "NUMBER PLEASE"
30 OUTPUT $ACCEPTNUM REQUEST
END
```

Now, for creation of conditional breaks.

```
TO $MAKECONDITIONAL
10 PRINT "CONDITION FOR BREAK..."
20 DO SENTENCES
     "30 IF" REQUEST "THEN BREAK"
30 DO "END"
END
```

The set of procedures for creating a breakpoint facility is now complete. The only things remaining are the bookkeeping procedures for keeping track of the breakpoints inserted. One of these, $ENTERBREAK is already mentioned in $INSBREAK2. It simply enters the breakname on the list "BREAK POINTS".

```
TO $ENTERBREAK
10 MAKE "BREAK POINTS" SENTENCES
     :BREAK POINTS: FIRST :PROC LINE:
          LAST :PROC LINE:
END
```

It is left to the reader to write the simple procedures which selectively or globally list and erase breakpoints.

4.2 Running a procedure over a specified input domain

In many circumstances it is desirable to run a student's programs in a mode different from their original operation. We next discuss sets of procedures to do this. Perhaps the very simplest generalization of simply trying a student's program with a sequence of different inputs is to provide many sets of input parameters at a time. The chief utility of this extension is that standard sets of inputs can be developed by the user and applied to student procedures very simply. The top-level procedure is $RUN, which takes three inputs -- the procedure to be exercised, the number of inputs, and the list of input sets. $RUN uses $PULLQ to pull and quote one set at a time and $CUT to give the remainder of the input set.

```
TO $RUN :PNAME: :# INPUTS: :INPUT LIST:
10 IF EMPTYP :INPUT LIST: STOP
20 DO SENTENCE :PNAME: $PULLQ :# INPUTS:
         :INPUT LIST:
30 $RUN :PNAME: $CUT :# INPUTS: :INPUT LIST:
END
```

```
TO $PULLQ :NUM: :LIST:
10 IF EMPTYP :LIST: EXIT "NOT ENOUGH INPUTS"
20 IF ZEROP :NUM: OUTPUT :EMPTY:
30 OUTPUT SENTENCES
         :QUOTE:
         FIRST :LIST:
         :QUOTE:
         $PULLQ (DIFF :NUM: 1) BUTFIRST :LIST:
END
```

```
TO $CUT :NUM: :LIST:
10 IF ZEROP :NUM: OUTPUT :LIST:
     ELSE OUTPUT $CUT (DIFF :NUM: 1)
     BUTFIRST :LIST:
END
```

Now, to show its use.

```
←TO FOO :A: :B:
@10 PRINT SUM OF :A: AND :B:
@END
FOO DEFINED
←$RUN "FOO" 2 "1 2 3 4 5 6 7 8 9 10"
3
7
11
15
19
←
```

It is easy to write a procedure for $RUN which computes the number of inputs of FOO, in fact, $COUNTARGS in the parsing section is just that procedure.

4.3 Running a procedure from a specified point

Another useful facility for debugging programs, especially those written by other people, is to run a program starting with some arbitrary line number. This is the case, for example, when one is confronted with a large (bad practice, of course) program whose initial part just generates a lot of printing. The procedure $RUNFROM :PNAME: :LINE: runs :PNAME: starting at :LINE: by inserting a GOTOLINE :LINE: as line 1 of :PNAME:. (If line 1 is already occupied, $RENUMBERing, discussed in Section 4.5, is called.) Line 2 is then defined to erase line 1, otherwise recursions might end badly. After :PNAME: has been executed, the two added lines 1, 2 are removed and the procedure is $UNRENUMBERED, if it was $RENUMBERED earlier.

```
TO $RUNFROM :PNAME: :LINE:
10 TEST $INTP "1 2" LINES :PNAME:
20 IFTRUE $RENUMBER :PNAME:
30 EDIT :PNAME:
40 IFTRUE DO SENTENCE "1 GOTOLINE" W :LINE: "0"
50 IFFALSE DO SENTENCE "1 GOTOLINE" :LINE:
60 DO "2 ERASE LINE 1"
70 DO "END"
80 DO :PNAME:
90 DO SENTENCE "EDIT" :PNAME:
100 ERASE LINE 1
110 ERASE LINE 2
120 DO "END"
130 IFTRUE $UNRENUMBER
END
```

```
TO $INTP :LIST 1: :LIST 2:
10 IF EMPTYP :LIST 1: OUTPUT "FALSE"
20 IF $MP FIRST :LIST 1: :LIST 2: OUTPUT "TRUE"
30 OUTPUT $INTP BUTFIRST :LIST 1: :LIST 2:
END
```

4.4 Testing of procedures which use random number generation

It is sometimes very difficult to track down bugs which turn up in procedures which use random variables; in LOGO these involve the built-in operation RANDOM. The bug may only exist for a very small fraction of values of a random variable, or the manifestation of the bug may vary widely in successive executions of the defective procedure. In using LOGO, there are several means at ones disposal for systematically varying RANDOM's outputs. The simplest, yet very effective, such method is to replace (by means of the user-defined $REPLACE, discussed in Section 4.5) each occurrence of RANDOM with a constant -- 0 being the best choice. The great success of this procedure is that the most common serious misuse of random numbers is forgetting that the value 0 can be assumed and therefore devising a defective end-test. By way of trivial example:

```
TO RANDOMCHOOSE :LIST:
10 OUTPUT CHOOSE RANDOM :LIST:
END
```

```
TO CHOOSE :N: :LIST:
10 IF (EQUALP :N: 1) OUTPUT FIRST OF :LIST:
20 OUTPUT CHOOSE (:N: - 1) BUTFIRST OF :LIST:
END
```

If RANDOM is replaced by 0 above, the otherwise intermittent bug is impaled. (Actually, it is better to $REPLACE RANDOM by 00000, say, which is numerically the same as 0, but easier to $UNREPLACE.)

For those very rare (yet very irritating) circumstances where the simple substitution described above doesn't work, a more methodical replacement of the random numbers is called for. To be able to do this, we must be able to repeat a procedure while systematically varying some of its internal parameters (as opposed to specifying an input domain as we did earlier in the simple case $RUN). Unfortunately, this is a very hard problem. One cannot simply systematically replace the, say 5, occurrences of RANDOM in the user procedures by registers which are then methodically "stepped through" from 0 to 9. Consider, for example, the following (rather poor) algorithm for generating quinary sequences of length :N: --

```
TO RANDOMQUINARY :N:
10 IF (EQUALP :N: 0) OUTPUT :EMPTY:
20 MAKE :DIGIT: RANDOM
30 TEST GREATERP 6 :DIGIT:
40 IFTRUE OUTPUT WORD
        :DIGIT:
        RANDOMQUINARY (DIFF :N: 1)
50 IFFALSE OUTPUT RANDOMQUINARY :N:
END
```

In this procedure, modified from actual student work, the RANDOM in line 20 is cycled through repeatedly in a single execution of RANDOMQUINARY, and very likely even more times through repeated calls on RANDOMQUINARY by higher level procedures. Furthermore, the number of times RANDOM is invoked in a single call to RANDOMQUINARY will vary with the values it assumes, from :N: to infinity, though, fortunately the probability of a given number of invocations falls rapidly as the number exceeds :N:.

It is clear from this simple example, together with even a slightly active imagination, that the general problem is pretty hard. So we try a new tack concentrating on the desired product, rather than the means. What we really want is a trace of the values assumed by RANDOM so that we can see which ones worked and which failed. This is easily done by replacing each occurrence of RANDOM with a procedure $RANDOM which, as well as printing the value that RANDOM assumes, also prints where it is -- this positional information is taken to be the input to $RANDOM so that a single version of this procedure suffices for all occurrences of RANDOM generation.

```
TO $RANDOM :PNAME LINENO:
10 PRINT SENTENCE "AT" :PNAME LINENO:
20 MAKE "PNAME LINENO" RANDOM
30 TYPE WORD "==>" :PNAME LINENO:
40 OUTPUT :PNAME LINENO:
END
```

In addition, each procedure containing occurrences of the $RANDOM procedure should be traced so that the aggregation of the more complex random quantities these may generate are clearly shown. To do all this we write a procedure which sweeps through any specified set of procedures, usually the entire :CONTENTS:, neglecting those procedures which begin with $. (For one thing we don't want to debug "system" procedures; for another, embarrassing things would happen -- to $RANDOM itself, for example.)

```
TO $BUGRANDOM :PLIST:                (sweeps through :PLIST:)
10 MAKE "PLIST" SENTENCES :PLIST:    (ensures that :PLIST: is a
                                      sentence)
20 IF EMPTYP :PLIST: STOP
30 TEST IS FIRST OF FIRST
     OF :PLIST: "$"
40 IFFALSE $BUGRANDOM1 FIRST :PLIST:
     LINES FIRST :PLIST:
50 $BUGRANDOM BUTFIRST :PLIST:
60 PRINT "FINISHED $BUGGING"
END
```

$BUGRANDOM1 will search for lines containing RANDOM. If one is found, it is suitably modified by $RANDOMIZE and the procedure containing the RANDOM is traced.

```
TO $BUGRANDOM1 :PNAME: :LINES:
10 IF EMPTYP :LINES: STOP
20 TEST $MP "RANDOM" TEXT :PNAME:
     FIRST :LINES:
30 IFTRUE DO SENTENCE "TRACE" :PNAME:
40 IFTRUE $RANDOMIZE TEXT :PNAME:
     FIRST :LINES:
50 $BUGRANDOM1 :PNAME: BUTFIRST :LINES:
END
```

```
TO $RANDOMIZE :LINE:
10 DO SENTENCE "EDIT" :PNAME:
20 DO SENTENCE FIRST :LINE: $REP
     BUTFIRST :LINE:
30 DO "END"
END
```

```
TO $REP :TEXT:                       (does the actual replacement)
10 IF EMPTYP :TEXT: OUTPUT :EMPTY:
20 TEST IS FIRST :TEXT: "RANDOM"
30 IFFALSE OUTPUT SENTENCE FIRST :TEXT:
     $REP BUTFIRST :TEXT:
40 IFTRUE OUTPUT SENTENCES
     "$RANDOM"
     :QUOTE:
     :PNAME:
     FIRST :LINE:
     :QUOTE:
     $REP BUTFIRST :TEXT:
END
```

To see how this set of five procedures works, let us try it out on the single procedure RANDOMQUINARY, defined earlier as an admittedly trivial example.

```
←$BUGRANDOM "RANDOMQUINARY"
FINISHED $BUGGING
←PRINT RANDOMQUINARY 2
RANDOMQUINARY OF 2
RANDOMQUINARY 20 ==> 6
  RANDOMQUINARY OF 2
RANDOMQUINARY 20 ==> 0
    RANDOMQUINARY OF 1
RANDOMQUINARY 20 ==> 9
      RANDOMQUINARY OF 1
RANDOMQUINARY 20 ==> 0
        RANDOMQUINARY OF 0
        RANDOMQUINARY OUTPUTS :EMPTY:
      RANDOMQUINARY OUTPUTS 0
    RANDOMQUINARY OUTPUTS 0
  RANDOMQUINARY OUTPUTS 00
RANDOMQUINARY OUTPUTS 00
00
←
```

A clear trace of all procedure operations relating to RANDOM is provided even in complex situations where several procedures use RANDOM and interact in nontrivial fashion. The reader is left the much easier task of $UNBUGRANDOMing, by writing the suitable set of procedures.

4.5 Editing Facilities - renumbering procedure lines

An ability to automatically renumber the lines of a program is a useful editing feature of any line oriented programming language. Although this facility is not included among the LOGO primitives, the primitives are easily extended to perform this function. To show the way in which editing commands can readily be added by an experienced user, we follow the development of a renumbering package in some detail.

When the procedure being modified is itself the object of inquiry, as is the case with dribble file analysis, it is desirable to also have facilities to undo changes. When one writes a procedure RENUMBER, a procedure UNRENUMBER is likely to be of additional use. Furthermore, a record should be automatically generated of which procedures have been modified by RENUMBER. Having specified his desired goals in this manner, the user now must cast about for a renumbering scheme which is invertible. The usual method (as in most flavors of BASIC) in which an initial number and step size are input parameters, requires that a separate record be kept of the original numbering, a clumsy and inefficient method.

Far simpler is the multiplication of each line number by a fixed constant. Division by that constant will then restore the line numbering to its original state. A choice of 10 provides adequate spacing in nearly all cases and results in a very transparent renumbering.

The careful user writing this renumbering package would also note two further points:

(A) The input of GOTOLINE must also be modified in each appearance of that command. This input need not be a number so insertion of an explicit PRODUCT (OF) 10 (AND) must be inserted.

(B) It matters in which order the lines of a procedure are renumbered. If one starts with the lowest number and works up, renumbering a given line will clobber a succeeding line if the two are in the ratio 1:10. A renumbering procedure starting with the highest number eliminates this difficulty (vice versa for unrenumbering).

Our hypothetical user follows the convention that "system" procedures are preceded by $ to avoid possible conflict with procedures defined by the dribble files themselves. He might start his procedure-writing, as is his usual style, from the top level down or from the bottom up, let us say the top down.

```
TO $RENUMBER :PNAME:
10 DO SENTENCE "EDIT" :PNAME:        (gets into redefinition mode)
20 DO SENTENCES
     "TITLE"                         (adds the comment "renumbered"
     BUTFIRST TEXT :PNAME: 0         to the title line of :PNAME:)
     ";RENUMBERED;"
30 $REN :PNAME:                      (renumbers each of the lines
     BUTFIRST LINES :PNAME:          of :PNAME: using--yet to be
                                     written--$REN)
40 DO "END"
50 PRINT SENTENCE :PNAME: "RENUMBERED"  (indicates that the
                                        renumbering is completed)
END
```

$REN will go through the list of line numbers, starting with the last one, creating a new copy of it with line number multiplied by ten, erase the old version and repeat with BUTLAST of the list till it is empty.

```
TO $REN :PNAME: :LIST:                (:LIST: is the list of line
                                      numbers to be modified)
10 IF EMPTYP :LIST: STOP
20 DO SENTENCE OF                     (creates a new line which has
     (WORD OF LAST :LIST: AND 0)      line number 10 times that of
   AND $CGOTO BUTFIRST TEXT*          the last one on the list and
       :PNAME: LAST :LIST:            has text modified by $CGOTO
                                      which looks for GOTO LINES
                                      and modifies them)
30 DO SENTENCE
     "ERASE LINE"                     (erases the old numbered
     LAST :LIST:                      version of the line)
40 $REN OF :PNAME: AND                (the last line number on
     BUTLAST OF :LIST:                :LIST: is taken care of, then
                                      the process is repeated with
END                                   the rest of the list)
```

Now to write $CGOTO, which replaces "GOTOLINE" by "GOTOLINE PRODUCT OF 10 AND", we use in turn the rather straightforward and generally useful procedure $REPLACE.

```
TO $CGOTO :TEXT:
10 $REPLACE "GOTOLINE" "GOTOLINE PRODUCT OF 10 AND"
     :TEXT:
END
```

```
TO $REPLACE :A: :B: :C:
10 IF EMPTYP :C: THEN OUTPUT :EMPTY:
20 IF IS FIRST :C: :A: THEN OUTPUT
     SENTENCE OF :B: AND BUTFIRST OF :C:
30 OUTPUT $REPLACE :A: :B: BUTFIRST :C:
END
```

Renumbering is now finished and we can create a dummy procedure FOO to test it out:

---

*Remember that TEXT :PROCEDURE NAME: :LINE NUMBER: gives the complete line including the line number.

```
←TO FOO :N:
@10 PRINT "I AM DOING FOO"
@20 GOTOLINE :N:
@END
FOO DEFINED
←$RENUMBER "FOO"
FOO RENUMBERED
←LIST FOO

TO FOO :N: ;RENUMBERED;
100 PRINT "I AM DOING FOO"
200 GOTOLINE PRODUCT OF 10 AND :N:
END

←
```

This is a little less than half of the goal we specified at the outset. Next we write the "unrenumbering" procedure. It is different from its inverse in that we check to see if the procedure :PNAME: has been renumbered and if not we abort the process.

```
TO $UNRENUMBER :PNAME:
10 TEST $MEMBERP ";RENUMBERED;"     (is the comment RENUMBERED on
     TEXT OF :PNAME: AND 0          the title line :PNAME:?)
20 IFFALSE PRINT SENTENCE :PNAME:   (if not, print message so
     "HAS NOT BEEN RENUMBERED"      indicating)
30 IFFALSE STOP                     (and stop)
40 DO SENTENCE "EDIT" :PNAME:       (get ready to edit :PNAME:)
50 DO SENTENCES
     "TITLE"
     REPLACE                        (remove ;RENUMBERED; from
        ";RENUMBERED;"              the title line of :PNAME:)
        "  "
        BUTFIRST TEXT :PNAME: 0
60 $UNREN :PNAME:                   (unrenumber the other lines
     BUTFIRST OF LINES OF :PNAME:   in :PNAME:)
70 DO "END"                         (leave editing mode)
80 PRINT SENTENCE :PNAME: "UNRENUMBERED"
END                                 (print terminating message)
```

$MEMBERP, which tests whether :ELEMENT: is a member of :LIST:, is rather straightforward.

```
TO $MEMBERP :ELEMENT: :LIST:
10 IF EMPTYP :LIST: OUTPUT "FALSE"
20 IF IS :ELEMENT: FIRST :LIST: OUTPUT "TRUE"
30 OUTPUT $MEMBERP :ELEMENT: BUTFIRST :LIST:
END
```

$UNREN is very much like its counterpart $REN, except that before dividing a line number by 10, it looks to see if the last digit is 0. If not, the line has certainly been added since renumbering was done and is ignored.

```
TO $UNREN :PNAME: :LIST:            (:LIST: is again the list of
                                    the line numbers)
10 IF EMPTYP :LIST: THEN STOP       (stop when :LIST: is exhausted)
20 TEST IS LAST OF FIRST OF :LIST: 0 (is the last digit of the
                                    first line number 0?)
30 IFFALSE $UNREN :PNAME:           (if not, go on to the next
     BUTFIRST OF :LIST:             line)
40 IFFALSE STOP
50 DO SENTENCE OF
     BUTLAST OF FIRST OF :LIST:     (divide line number by 10)
     $UNCGOTO OF BUTFIRST OF TEXT
         :PNAME:                    (we again have to deal with
         FIRST :LIST:               GOTOs)
60 DO SENTENCE "ERASE LINE"         (erase the un-unrenumbered
     FIRST :LIST:                   version of the line)
70 $UNREN :PNAME: BUTFIRST :LIST:   (repeat for the rest of
                                    :LIST:)
END
```

$UNCGOTO is a little more complicated than $CGOTO since a sentence rather than a word is searched for. Rather than use a more general $REPLACE, we write $UNCGOTO in one piece without using $REPLACE.

```
TO $UNCGOTO :TEST:
10 IF EMPTYP :TEXT: THEN OUTPUT :EMPTY:
20 TEST IS FIRST OF :TEXT: "GOTOLINE"
30 IFTRUE TEST IS $FIRST4 OF BUTFIRST  ($FIRST4 gives the first
     :TEXT: :LIST:                     four elements)
     "PRODUCT OF 10 AND"
40 IFTRUE OUTPUT                       ($BUTFIRST5 removes the
     $BUTFIRSTS5 OF :TEXT:             first five elements)
50 OUTPUT SENTENCE FIRST :TEXT:
     $UNCGOTO OF BUTFIRST :TEXT:
END
```

(We note, in passing, that $FIRST4 and $BUTFIRST5 can be better written as special cases of more general procedures $FIRSTN "4" and $BUTFIRSTN "5".)

Now we have written all the procedures to undo renumbering; we test them by returning to our dummy procedure FOO.

```
←$UNRENUMBER "FOO"
FOO UNRENUMBERED
←LIST FOO

TO FOO :N:
10 PRINT "I AM DOING FOO"
20 GOTOLINE :N:
END
```

Thus the original version of FOO is restored. This unrenumbering process can only be done once. Repetitions will be ineffective.

```
←$UNRENUMBER "FOO"
FOO HAS NOT BEEN RENUMBERED
←
```

The only work remaining is to obtain from the set of procedures defined (:CONTENTS:) that subset which has been renumbered.

```
TO $SEARCHREN
10 OUTPUT $SRC :CONTENTS:            (we must initialize our
END                                  search by specifying a list
                                     of procedure names)

TO $SRC :PLIST:
10 IF EMPTYP :PLIST: THEN OUTPUT
     :EMPTY:
20 TEST MEMBERP ";RENUMBERED;"       (test if the first procedure
     TEXT FIRST :PLIST: "0"          in :PLIST: has been renumbered)
30 IFTRUE OUTPUT SENTENCE
     FIRST :PLIST:                   (if so, include it in the
     $SRC BUTFIRST OF :PLIST:        output and go on)
40 OUTPUT $SRC BUTFIRST :PLIST:      (if not, just go on)
END
```

And, we can even write a procedure which unrenumbers everything.

```
TO $UNRENUMBERALL
10 $UNRALL $SEARCHREN                (initialization)
END

TO $UNRALL :LIST:
10 IF EMPTYP :LIST: THEN STOP
20 $UNRENUMBER FIRST OF :LIST:
30 $UNRALL BUTFIRST OF :LIST:
END
```

This is a fairly substantial, sophisticated product for a relatively small, unsophisticated amount of work. Such a package should be the product of no more than two hours of work by a reasonably experienced LOGO user of average ability.

4.6 Further examples of editing aids

In the preceding section we developed one particular aid -- a renumbering package -- in some detail. We did so for purposes of illustration - not all users will feel a need for such a capability and those who do may have differing views on the form and effects of such a package. In this section we briefly discuss some further "stand alone" aids which can easily be written to augment editing capabilities.

Modifying some or all of the procedures in one's workspace is often useful, as for example, when the name of a procedure is to be consistently changed. This is easily done by using DO, LINES, and TEXT in much the same way as in the renumbering example. The form of the top-level procedure might be

REPLACE (old text) (new text) (procedure list)

where the procedure list specifies those procedures in which the substitution is to be made. (:CONTENTS: can be used to specify all procedures.) An example of a REPLACE procedure similar to this is developed in Appendix 1.

A related idea is to develop a procedure

FIND (text) (procedure list) which simply enumerates the occurrences of (TEXT) in the domain specified by (procedure list).

Both of the above types of procedure can be much enhanced by the use of the set of pattern matching procedures which we have written in LOGO.[1] They provide a capability much the same as that in SNOBOL2. These pattern matching programs are accessed by a top-level procedure MATCH (pattern list) (text). MATCH is a predicate which returns true or false as it succeeds or fails. A pattern list is a string of variables and literal text. In the case of success, any variables used in specification of the pattern are set to those values which resulted in success.

---

[1] These are described in "Uses of the LOGO Programming Language in Undergraduate Instruction" Lukas, George, Proceedings National Conference, Association for Computing Machines, 1971.

Variables are distinguished from literal text to be matched by their first character being $. They may be followed by a pair of parentheses enclosing a type specification for the variable. The types of variable currently "built-in" include

PAR	must have correctly matching parentheses
NUM	must be numerical
(number)	must be of length (number)

The matching package also provides the user an area to insert his own variable specifications.

Examples of the use of MATCH are:

```
←PRINT MATCH "AB" "ABC"
FALSE
←PRINT MATCH "A $U" "ABC"
TRUE
←PRINT :$U:
BC
←PRINT MATCH "$U(PAR) * $V(PAR)" "(X+2)*(X+3)"
TRUE
←PRINT :$U:
(X+2)
←PRINT :$V:
(X+3)
```

Two notes should be made regarding the use of variables in pattern specification. First, the use of repeated matching occurrences of a variable in a pattern results in the requirement that all occurrences be identical, unlike SNOBOL2. Second, declaration of type of variable need only be made for one occurrence of the variable in a pattern list, although the declaration can be repeated if desired.

The pattern matching capabilities can be used to advantage in all phases of the analysis. Thus, when working directly with the student programs, matching procedures can be used to locate and correct bugs of specified form; for example, to replace all occurrences of the form PROCEDURENAME (A,B,C) by PROCEDURENAME (B,A,C). The entire parsing package can be managed by a pattern-matching-based executive program. It can, for example, make the parenthesis checking procedures more powerful by extracting the content for further analysis. It can easily separate out the student comments and do specified keyword searches on them to help guide the analyst's run time work.

4.7 Inserting Comments into the Procedure Structure Diagram

The "empty" procedure $NOTE is provided in the graphics portion of the RUN package to enable comments to be displayed on the procedure structure diagram along with the procedure name. $NOTE simply gives the transformation of its input desired on the diagram. Thus, initially $NOTE is defined:

```
TO $NOTE :PNAME:
10 OUTPUT :PNAME:
END
```

Let us say, for example, that a predicate $RECURSEP :PNAME: has been defined by the user, which outputs "TRUE" or "FALSE" as its input, :PNAME:, is or is not recursive. $NOTE can then be used to indicate recursiveness on the procedure structure diagram by "starring" recursive procedures. This is done simply by the addition of a line to its definition

```
5 IF $RECURSEP :PNAME: OUTPUT WORDS "**" :PNAME: "**"
```

This facility is used in Example 4 of Part 4.

Part 4.

Examples of System Use

1. Introduction

The dribble file analysis system has been described and documented in the preceding parts of this report. In the following pages we illustrate the use of the system in concrete applications. Each application emphasizes a particular aspect of the system's use. The first example shows how the standard facilities of the system are used in routine inspection of a student's work. It illustrates the various commands for executing dribble file lines and shows how procedures are listed and diagrammed during the course of running through the dribble file. The second example is an analytic study of a student's work during an extensive program debugging session. It shows how the system can provide the analyst with very specific and detailed insights about the student's difficulties, cognitive style, and current progress.

The third and fourth examples illustrate features useful for analyzing relatively complex program structures. The particular program structures we have chosen for illustration deal with random generation of grammatic sentences, and automatic extrapolation of number sequences. Both are straightforwardly written, compact, and easily understood. Nevertheless, they comprise a number of component programs, approximately 10 in each case, interconnected at approximately four levels of depth, and some of the programs are recursive. In the third example we illustrate the use of the procedure diagramming facility for graphically displaying such complex structures in a fairly transparent manner.

In the last example we illustrate the use of analyst-written procedures to assist in characterizing complex structures of this kind and also in actually augmenting the built-in procedure diagramming facility to indicate which procedures are simply recursive.

These examples are treated in the next section. In each case, discussions accompany the interactions made using the dribble file analysis system. The interactions were recorded using the photocopy device associated with the IMLAC display scope. These pictures form the basis for the discussions. We have typically combined two or three such scope photographs into each figure so as to make the presentations more concise.

2. Example 1

The dribble files used in this example and the following one were generated by University of Massachusetts undergraduates in a remedial computer mathematics course. Both examples are drawn from student work in geometry. The first example mainly concerns the development of a procedure for drawing triangles. As will be evident from looking at the student's work, he did not find this to be a trivial task.

Figure 1 shows the beginning of the analysis. In the first line, the user starts the analysis system by typing $STARTRUN. On line 2, the system requests the name of the dribble file to be analyzed (DRIBBLE FILE:); the user responds by typing the name of the lesson (LESSON PARSED). Then, on line 3, he calls for execution of the first lines of the dribble file, up to line 15, ($DOTO 15). Dribble file lines ***1*** through ***6*** are executed with no difficulty -- these constitute the definition of the procedure TRIANGLE. Line ***7*** however, where the student had called for execution of this procedure with an input of 3, ran into a problem, causing LOGO to stop. The diagnostic states the student's error (MARK NEEDS A MEANING) and indicates where the error occurred (I WAS AT LINE 10 IN TRIANGLE). At this point the analysis system has stopped and waits for the user's next command.

During the course of execution of dribble file lines, the procedure currently being defined, if any, is displayed in its current form in half-sized text at the lower left corner of the display. Note in Figure 1 that TRIANGLE is so displayed. Because the small text characters are difficult to read on the photocopy (though not, of course, on the face of the scope), we have shown the definition in standard size at the lower right corner of the figure (and we follow this convention in subsequent figures).

In Figure 2 the user proceeds with the command $DOTO 25. The system then executes dribble file lines ***9*** through ***25*** without being halted. (These lines define the procedures MARK, SUPERMARK, and RECTANGLE.) At this point the user types $WHERE and the system responds AT DRIBBLE LINE 25 WHICH IS END and then names the procedures which have thus far been defined. (Note that at this point, the procedure which has most currently been defined, RECTANGLE, is shown at the lower left corner.) The user then types the command $DOALL, which calls for the execution of the remainder of the dribble file.

As shown in Figure 3, the system is only able to execute the lines up to ***61***. There it ran into trouble, so indicated, (THERE ARE 1 INPUTS MISSING FOR MARK. I WAS AT LINE 10 IN TRIANGLE) and stopped. At this point the user listed the student's procedures TRIANGLE and MARK to look at their current definitions. Next (Figure 4) he proceeds with another $DOALL. This time the system stops with an error indication after line ***66***.

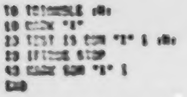
The user proceeds in this fashion through line ***78*** where another error stop occurs (TRIANGLE HAS NOT BEEN COMPLETELY DEFINED) where the student had attempted to execute TRIANGLE.

Subsequently (Figure 5), following the $WHERE command, the user calls for a diagram of the procedure STRIPE by typing $DISPLAY "STRIPE". The procedure structure diagram for STRIPE is shown. (As with the display of current definitions, small text is used in these diagrams also. To aid the reader, the names of the procedures displayed in the boxes are typed in at the right of the diagram.) The diagram shows that STRIPE uses SUBRECTANGLE which, in turn, uses SUPERMARK and RECTANGLE. Also, that SUPERMARK uses MARK and that RECTANGLE uses SUPERMARK. More information about diagrams is given in the section discussing Example 3.

As seen in Figure 6, the user also diagrams the procedure TRIANGLE, which has a simple procedure structure (it uses the two procedures NUM and MARK neither of which itself uses other procedures). This diagram is shown at the bottom of the figure. After listing the procedures NUM and TRIANGLE, he calls for the execution of the remainder of the dribble file with a $DOALL. Completion is indicated by the system printing ****END-OF-FILE****. The user then types PRINT :FINAL CONTENTS: to list the student procedures extant at the end of the run.

The student work in this dribble file consisted almost entirely in an attempt to write a good TRIANGLE procedure. He did not succeed, despite an extensive (but not insightful) editing effort. Along the way he wrote over 20 distinct versions of TRIANGLE. The $STEPPROCEDURE facility, in cases like this, gives a thumbnail history of the progressive stages in the development of a program. The analyst requests an enumeration of all these versions by typing $STEPPROCEDURE "TRIANGLE". The system responds (Figures 7, 8, and 9) with a listing of these successive definitions in the order in which the student had created them. In each new version, changed lines are set off with the prefix =>.

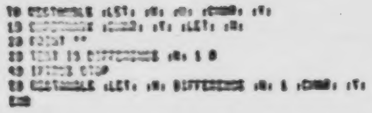
```
←$STARTRUN
DRIBBLE FILE:LESSON PARSED
←@DOTO 15
*** 1 *** TO TRIANGLE : N :
*** 2 *** 10 MARK " X "
*** 3 *** 20 T IS SUM " X " 1 : N :
*** 4 *** 30 IFT STOP
*** 5 *** 40 MARK SUM " X " 1
*** 6 *** END
*** 7 *** TRIANGLE 3
MARK NEEDS A MEANING.
I WAS AT LINE 10 IN TRIANGLE
←
```



```
TO TRIANGLE :N:
10 MARK "X"
20 TEST IS SUM "X" 1 :N:
30 IFTRUE STOP
40 MARK SUM "X" 1
END
```

Figure 1.

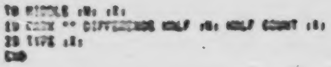
```
←*DOTO  25
*** 9 *** TO MARK : CHAR : : N :
*** 10 *** 10 TYPE : CHAR :
*** 11 *** 20 TEST IS DIFFERENCE : N : " 1 " " 0 "
*** 12 *** 30 IFTRUE STOP
*** 13 *** 40 MARK : CHAR : DIFFERENCE : N : " 1 "
*** 14 *** END
*** 15 *** TO SUPERMARK : CHAR : : N : : LET : : M :
*** 16 *** 10 MARK : CHAR : : N :
*** 17 *** 20 MARK : LET : : M :
*** 18 *** END
*** 19 *** TO RECTANGLE : LET : : M : : N : : CHAR : : Y :
*** 20 *** 10 SUPERMARK : CHAR : : Y : : LET : : M :
*** 21 *** 20 PRINT " "
*** 22 *** 30 TEST IS DIFFERENCE : N : " 1 " " 0 "
*** 23 *** 40 IFTRUE STOP
*** 24 *** 50 RECTANGLE : LET : : M : DIFFERENCE : N : " 1 " : CHAR : :
Y :
*** 25 *** END
←*WHERE
AT DRIBBLE LINE 25
     WHICH IS
END
THE PROCEDURES TRIANGLE MARK SUPERMARK RECTANGLE HAVE BEEN DEFINED
←*DDALL
```



```
TO RECTANGLE :LET: :M: :N: :CHAR: :Y:
10 SUPERMARK :CHAR: :Y: :LET: :M:
20 PRINT ""
30 TEST IS DIFFERENCE :N: "1" "0"
40 IFTRUE STOP
50 RECTANGLE :LET: :M: DIFFERENCE :N: "1"
              :CHAR: :Y:
END
```

Figure 2.

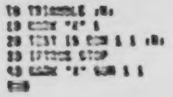
```
### 46 ### 30 OUTPUT DELETE BUTFIRST : CHAR : DIFFERENCE : N : " 1 "
### 47 ### END
### 48 ### TO STRIPE : M : : N : : Y : : S :
### 49 ### 1 SUBRECTANGLE : M : : N : : Y :
### 50 ### 20 SUBRECTANGLE : M : : N : SUM : Y : " 1 "
### 51 ### 30 TEST IS SUM : Y : " 1 " : S :
### 52 ### 40 IFTRUE STOP
### 53 ### 50 STRIPE : M : : N : SUM : Y : " 1 " : S :
### 54 ### END
### 55 ### TO MIDDLE : N : : X :
### 56 ### 10 MARK " " DIFFERENCE HALF : N : HALF COUNT : X :
### 57 ### 20 TYPE : X :
### 58 ### END
### 61 ### TRIANGLE 3
THERE ARE 1 INPUTS MISSING FOR MARK.
I WAS AT LINE 10 IN TRIANGLE

←LIST TRIANGLE

TO TRIANGLE :N:
10 MARK "X"
20 TEST IS SUM "X" 1 :N:
30 IFTRUE STOP
40 MARK SUM "X" I
END

←LIST MARK

TO MARK :CHAR: :N:
10 TYPE :CHAR:
20 TEST IS DIFFERENCE :N: 1 0
30 IFTRUE STOP
40 MARK :CHAR: DIFFERENCE :N: 1
END
```



```
TO MIDDLE :N: :X:
10 MARK "" DIFFERENCE HALF :N:
        HALF COUNT :X:
20 TYPE :X:
END
```

Figure 3.

```
←#DOALL
*** 62 *** EDIT TRIANGLE
*** 63 *** 10 MARK " X " 1
*** 64 *** 40 MARK " X " SUM " X " 1
*** 65 *** END
*** 66 *** TRIANGLE 3
X
SUM OF "X" AND "1"
INPUTS MUST BE NUMBERS.
I WAS AT LINE 20 IN TRIANGLE

    : : :



*** 75 *** LIST SUPERMARK

TO SUPERMARK :CHAR: :N: :LET: :M:
10 MARK :CHAR: :N:
20 MARK :LET: :M:
END

*** 76 *** EDIT TRIANGLE
*** 71 *** TRIANGLE : N : : M :
TRIANGLE HAS NOT BEEN COMPLETELY DEFINED.
```



```
TO TRIANGLE :N:
10 MARK :X: 1
20 TEST IS SUM 1 1 :N:
30 IFTRUE STOP
40 MARK "X" SUM 1 1
END
```

Figure 4.

```
←@WHERE
AT DRIBBLE LINE 97
     WHICH IS
TRINGLE 4
THE PROCEDURES TRIANGLE MARK SUPERMARK RECTANGLE SUBRECTANGLE FIND HALF
DELETE STRIPE MIDDLE NUM TRIANGLE HAVE BEEN DEFINED
←@DISPLAY "STRIPE"
←.
```

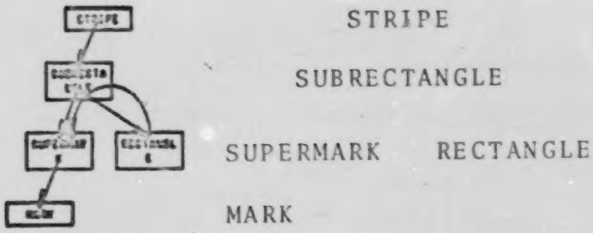

STRIPE
SUBRECTANGLE
SUPERMARK
RECTANGLE
MARK

Figure 5.

```
←*DISPLAY "TRIANGLE"
←LIST NUM

TO NUM :N:
10 MAKE :N: 1
END

←LIST TRIANGLE

TO TRIANGLE :M:
10 MARK "X" NUM :N:
20 PRINT ""
30 TEST IS DIFFERENCE :M: 1 0
40 IFTRUE STOP
50 MARK "X" SUM NUM :N: 1
END


←*DOALL
*** 187 *** EDIT TRIANGLE
*** 188 ***
****END-OF-FILE****


←PRINT :FINAL CONTENTS:
TRIANGLE 1 1 MARK 9 2 SUPERMARK 15 4 RECTANGLE 19 5 SUBRECTANGLE 26 3
FIND 33 1 HALF 39 1 DELETE 43 2 STRIPE 48 4 MIDDLE 55 2 NUM 87 1
TRIANGLE 90 1
```

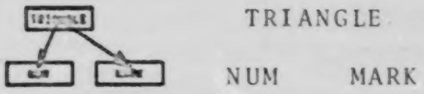


Figure 6.

```
←*STEPPROCEDURE "TRIANGLE"
==>TO TRIANGLE : N :
*
TO TRIANGLE : N :
==>10 MARK " X "
*
TO TRIANGLE : N :
10 MARK " X "
==>20 T IS SUM " X " 1 : N :
*
TO TRIANGLE : N :
10 MARK " X "
20 T IS SUM " X " 1 : N :
==>30 IFT STOP
*
TO TRIANGLE : N :
10 MARK " X "
20 T IS SUM " X " 1 : N :
30 IFT STOP
==>40 MARK SUM " X " I
*
TO TRIANGLE : N :
==>10 MARK " X " 1
20 T IS SUM " X " 1 : N :
30 IFT STOP
40 MARK SUM " X " I
*
TO TRIANGLE : N :
10 MARK " X " 1
20 T IS SUM " X " 1 : N :
30 IFT STOP
==>40 MARK " X " SUM " X " 1
*
TO TRIANGLE : N :
10 MARK " X " 1
==>20 T IS SUM 1 1 : N :
30 IFT STOP
40 MARK " X " SUM " X " 1
*
TO TRIANGLE : N :
10 MARK " X " 1
20 T IS SUM 1 1 : N :
30 IFT STOP
==>40 MARK " X " SUM 1 1
*

:  :  :
```

Figure 7.

```
TO TRIANGLE : M :
10 MARK " X " NUM : N :
20 PRINT " "
==>30 T IS DIFF : M : 1 0
*
TO TRIANGLE : M :
10 MARK " X " NUM : N :
20 PRINT " "
30 T IS DIFF : M : 1 0
==>40 IFT STOP
*
TO TRIANGLE : M :
10 MARK " X " NUM : N :
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
==>50 MARK " X " SUM NUM : N : 1
*
TO TRIANGLE : M :
==>10 MARK " X " " N "
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MARK " X " SUM NUM : N : 1
*
TO TRIANGLE : M :
==>10 MARK " X " " NU "
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MARK " X " SUM NUM : N : 1
*
TO TRIANGLE : M :
10 MARK " X " " NU "
==>20 MARK " X " SUM " NU " 1
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MARK " X " SUM NUM : N : 1
*
TO TRIANGLE : M :
10 MARK " X " " NU "
==>20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MARK " X " SUM NUM : N : 1
*

.  .  .
.  .  .
.  .  .
```

Figure 8.

```
TO TRIANGLE : M :
==>10 MARK " X " : NU :
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MARK " X " SUM " NU " 1
*
TO TRIANGLE : M :
10 MARK " X " : NU :
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
==>50 MARK " X " SUM : NU : 1
*
TO TRIANGLE : M :
10 MARK " X " : NU :
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
==>50 MAKE
*
TO TRIANGLE : M :
10 MARK " X " : NU :
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MAKE
==>60 TRIANGLE : M :
*
TO TRIANGLE : M :
10 MARK " X " : NU :
20 PRINT " "
30 T IS DIFF : M : 1 0
40 IFT STOP
50 MAKE
==>60 TRIANGLE : N :
*
TO TRIANGLE : M :
10 MARK " X " : NU :
20 PRINT " "
==>30 T IS DIFF : N : 1 0
40 IFT STOP
50 MAKE
60 TRIANGLE : N :
*.
```

Figure 9.

3. Example 2

This example shows the use of the system in an intensive, deep, sustained analysis of student work at operational and intentional levels. The student's work shown in this dribble file is aimed at creating a procedure DIAMOND for drawing a diamond-shaped figure. The student's plan in designing this procedure is to create two sub-procedures -- NUM for drawing triangle and UPD for drawing an "upside-down" triangle. The execution of a NUM followed by the execution of an UPD with matching input should produce the desired result. Figure 10 shows the analyst's execution of the initial lines of the student's dribble file, via the command $DOALL. These lines define the procedures MARK, SUPERMARK, RECTANGLE, SUBRECTANGLE, and (partially) FIND. A little later on, as seen at the beginning of Figure 11, the procedure NUM appears to be working. NUM simply initializes and invokes the triangle drawing procedure TRIANGLE. Student line ***69*** is an execution of NUM 4 and this results in the drawing of a triangle with 4 rows of X's. Already the student's work seems to be half finished.

In dribble file lines ***70*** through ***78*** the student has defined the upside down triangle drawing procedure UPD. On line ***79*** he has called for the execution of the procedure UPD4. This is an error (he meant to write UPD 4). So LOGO complains that UPD4 NEEDS A MEANING. At this point the analyst executes a $WHERE to list the currently defined procedures. He then diagrams the procedure structure of TRIANGLE (Figure 12) and proceeds by executing the next lines of the student's program with $DOALL (Figure 13).

Now the student has correctly called for the execution of his UPD procedure (Line ***80***). But, UPD 4 does not produce the desired upside down triangle. Instead, it continues indefinitely to draw rows of 4 X's. The analyst terminates this with a BREAK and then lists the procedure UPD. Superficially it appears correct -- it has a stopping condition and end test defined in lines 30 and 40 and a decremental iteration of the input in line 50. (This directs it to draw two less X's on each successive row of the upside down triangle.) But, obviously something is wrong.

Figure 14 shows the analyst executing the next lines of the student's work. The student has started to debug UPD. In lines ***82*** he puts a TRACE on UPD and then executes UPD 4 again. The trace lists the successive invocations of UPD. The correct sequence of calls should begin UPD OF "4", UPD OF "2", ... Instead, UPD OF "4" calls UPD OF "4" indefinitely. After a BREAK the analyst executes another $DOALL to see the student's next move. The student has now decided to list UPD. After this (line ***86***) he once more executes UPD 4. (Probably he couldn't see that anything was wrong and wanted to try the procedure again -- perhaps the computer had made an error of some kind.) But this produces the same unfortunate result.

By the next line, some light has dawned. Figure 15 shows the student fixing a bug in UPD. Dribble file lines ***91*** through ***93*** show him editing the procedure. He changes line 50 of his procedure from

```
MAKE :N: DIFF :N: 2
```

to

```
MAKE "N" DIFF :N: 2
```

This makes effective the decrementing of :N: by 2 on each round. The student calls UPD 4 again on the following line. And now another problem appears: UPD 4 writes a row of four X's and then calls UPD 2 which writes a row of two X's, which in turn calls UPD 0 which writes what appears to be an endless row of X's. The analyst breaks the execution of UPD.

The following $DOALL exposes the student's next line. He executes UPD 0 to confirm its nonterminating effect. Then (in line ***99*** in Figure 16) he has traced this effect down one level to find the subprocedure MIDDLE responsible. MIDDLE 50 1 0 produces the same nonterminating sequence of marks (1's in this case). At this point the analyst lists the procedure MIDDLE to see what *it* does. And, as is shown, MIDDLE invokes the subprocedure MARK two times. He then executes the student's next line ***101*** which shows the student himself running the subprocedure MARK and observing that MARK 1 0 replicates the results of MIDDLE 50 1 0 and UPD 0, its big brothers.

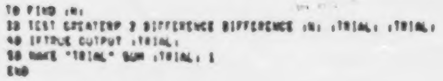
After breaking the execution of MARK, the analyst lists MARK :CHAR: :N: and presumably sees that, when its second input is 0, MARK will indeed fail to stop. Instead it will slip through the test for :N:=0 and indefinitely continue with a sequence of negative :N: values. This is also understood by the student who has (Figure 17) traced MARK and certainly noted this. The student's response is interesting. Instead of debugging MARK so that its stopping condition will work for even :N: as well as for odd :N:, he has evidently realized that his DIAMOND procedure only invokes MARK with odd values of :N: (since diamonds always have odd numbers of X's in their rows; thus UPD 1 will make the last call to MARK and MARK "X" 1 will stop after typing a single "X" mark). So the student realizes that he can ignore the difficulty with MARK, since it is not relevant to his goal, and he proceeds with DIAMOND. In lines ***112*** through ***115*** he defines DIAMOND as NUM 8 followed by UPD 15.

Then (Figure 15) he executes this DIAMOND procedure and it works. Obviously, though, he is unsatisfied with it. It is only capable of drawing the single diamond made up of an 8-rowed triangle on top of an 8-rowed upside down triangle. In the next few lines he erases this limited drawing procedure and defines a more general DIAMOND procedure with two inputs. DIAMOND :L: :V: is defined as NUM :L: followed by UPD :V:. He tries this (Figure 19) with a 9-rowed triangle and it works at once.

But, he is still dissatisfied with the inelegance of the definition. The procedure should be smart enough to work with a single input and automatically match the interface between the triangle made by NUM and the upside down triangle made by UPD. So he again erases the current version of DIAMOND and rewrites it as a single-input procedure DIAMOND :L: composed of NUM :L: and UPD :L: + :L: -1. Upon trying DIAMOND 5 he finds that the seams do not quite match -- the first row of X's made by UPD is the same size as the last row of X's made by NUM (instead of two X's smaller).

He fixes his procedure (Figure 20) by changing the input of UPD to :L: + :L: -3. Now his procedure apparently satisfies his goal. He exercises it several times with various inputs, including (line ***151***) a random input. After listing his procedures MIDDLE and TRIANGLE and asking for the time of day, he starts to log out. In Figure 21 the analyst executes a $WHERE, displays the procedure structure diagram for DIAMOND, and executes the last lines of the dribble file.

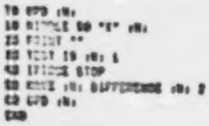
```
#DOALL
*** 3 *** TO MARK : CHAR : : N :
*** 4 *** 10 TYPE : CHAR :
*** 5 *** 20 TEST IS DIFFERENCE : N : " 1 " " 0 "
*** 6 *** 30 IFTRUE STOP
*** 7 *** 40 MARK : CHAR : DIFFERENCE : N : " 1 "
*** 8 *** END
*** 9 *** TO SUPERMARK : CHAR : : N : : LET : : M :
*** 10 *** 10 MARK : CHAR : : N :
*** 11 *** 20 MARK : LET : : M :
*** 12 *** END
*** 13 *** TO RECTANGLE : LET : : M : : N : : CHAR : : Y :
*** 14 *** 10 SUPERMARK : CHAR : : Y : : LET : : M :
*** 15 *** 20 PRINT " "
*** 16 *** 30 TEST IS DIFFERENCE : N : " 1 " " 0 "
*** 17 *** 40 IFTRUE STOP
*** 18 *** 50 RECTANGLE : LET : : M : DIFFERENCE : N : " 1 " : CHAR : :
Y :
*** 19 *** END
*** 20 *** TO SUBRECTANGLE : M : : N : : Y :
*** 21 *** 10 SUPERMARK "   " : Y : " X " : M :
*** 22 *** 15 PRINT " "
*** 23 *** 20 TEST IS DIFFERENCE : N : " 1 " " 0 "
*** 24 *** 30 IFTRUE STOP
*** 25 *** 40 RECTANGLE " X " : M : DIFFERENCE : N : " 1 " "   " : Y :
*** 26 *** END
*** 27 *** TO FIND : N :
*** 28 *** 30 TEST GREATERP " 2 " DIFFERENCE DIFFERENCE : N : : TRIAL :
: TRIAL :
*** 29 *** 40 IFTRUE OUTPUT : TRIAL :
*** 30 *** 50 MAKE " TRIAL " SUM : TRIAL : " 1 "
*** 31 *** 60 OUTPUT FIND : N :
```



```
TO FIND :N:
30 TEST GREATERP 2 DIFFERENCE
   DIFFERENCE :N: :TRIAL: :TRIAL:
40 IFTRUE OUTPUT :TRIAL:
50 MAKE "TRIAL" SUM :TRIAL: 1
END
```

Figure 10.

```
*** 66 *** EDIT TRIANGLE
*** 67 *** 30 T IS SUM : N : DIFF : N : 1 : NU :
*** 68 *** END
*** 69 *** NUM 4
                       X
                      XXX
                     XXXXX
                    XXXXXXX
*** 70 *** TO UPD : N :
*** 71 *** 10 MIDDLE 50 " X " : N :
*** 72 *** 20 PRINT " "
*** 73 *** 30 T IS : N : 1
*** 74 *** IFT STOP
*** 75 *** 40 IFT STOP
*** 76 *** 50 MAKE : N : DIFF : N : 2
*** 77 *** 60 UPD : N :
*** 78 *** END
*** 79 *** UPD4
UPD4 NEEDS A MEANING.
I WAS AT LINE 35 IN *DOLINE
```



```
TO UPD :N:
10 MIDDLE 50 "X" :N:
20 PRINT ""
30 TEST IS :N: 1
40 IFTRUE STOP
50 MAKE :N: DIFFERENCE :N: 2
60 UPD :N:
END
```

Figure 11.

```
←#WHERE
AT DRIBBLE LINE 79
     WHICH IS
UPD4
THE PROCEDURES MARK SUPERMARK RECTANGLE SUBRECTANGLE FIND HALF DELETE
STRIPE MIDDLE NUM TRIANGLE UPD HAVE BEEN DEFINED
←#DISPLAY "TRIANGLE"
←
```

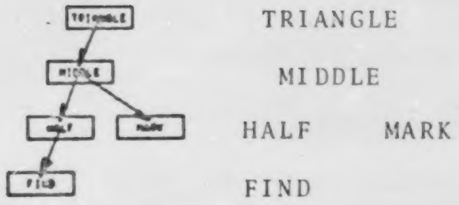

TRIANGLE
MIDDLE
HALF
MARK
FIND

Figure 12.

```
←*DOALL
*** 80 *** UPD 4
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
                                    XXXX
BREAK
I WAS AT LINE 40 IN FIND
←LIST UPD

TO UPD :N:
10 MIDDLE 50 "X" :N:
20 PRINT ""
30 TEST IS :N: 1
40 IFTRUE STOP
50 MAKE :N: DIFFERENCE :N: 2
60 UPD :N:
END

←
```

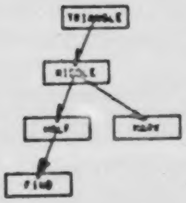

TRIANGLE
MIDDLE
HALF
MARK
FIND

Figure 13.

```
←*DOALL
*** 81 *** TRACE UPD
*** 82 *** UPD 4
UPD OF "4"
            XXXX
 UPD OF "4"
            XXXX
  UPD OF "4"
            XXXX
   UPD OF "4"
            XXXX
    UPD OF "4"
            XXXX
     UPD OF "4"

BREAK
I WAS AT LINE 10 IN MARK

←*DOALL
*** 84 *** LIST UPD

(TRACED) TO UPD :N:
10 MIDDLE 50 "X" :N:
20 PRINT ""
30 TEST IS :N: 1
40 IFTRUE STOP
50 MAKE :N: DIFFERENCE :N: 2
60 UPD :N:
END

*** 86 *** UPD 4
UPD OF "4"
            XXXX
 UPD OF "4"
            ....
  UPD OF "4"
            ....
   UPD OF "4"
            ....
```

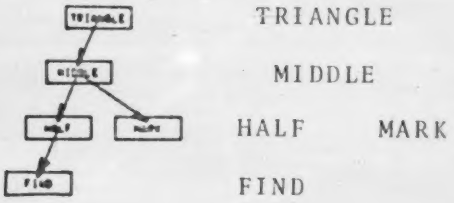
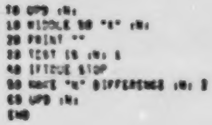

TRIANGLE
MIDDLE
HALF
MARK
FIND
TRIANGLE
MIDDLE
HALF
MARK
FIND

Figure 14.

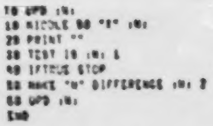
```
*** 88 *** CANCEL
CANCEL NEEDS A MEANING.
I WAS AT LINE 35 IN #DOLINE
←#DOALL
*** 89 *** P : 4 :
2
*** 90 *** P : 6 :

*** 91 *** EDIT UPD
*** 92 *** 50 MAKE " N " DIFF : N : 2
*** 93 *** END
*** 94 *** UPD 4
UPD OF "4"
                      XXXX
  UPD OF "2"
                       XX
    UPD OF "0"
                        ----------------------------------------------
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXX
BREAK

←#DOALL
*** 97 *** UPD 0
UPD OF "0"
                        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
BREAK
I WAS AT LINE 30 IN MARK
←
```



```
TO UPD :N:
10 MIDDLE 50 "X" :N:
20 PRINT ""
30 TEST IS :N: 1
40 IFTRUE STOP
50 MAKE "N" DIFFERENCE :N: 2
60 UPD :N:
END
```

Figure 15.

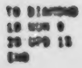
```
←#DOALL
*** 99 *** MIDDLE 50 1 0
                         11111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111
BREAK
I WAS AT LINE 20 IN MARK
←LIST MIDLE
←LIST MIDDLE

TO MIDDLE :N: :X: :Y:
10 MARK "*" DIFFERENCE HALF :N: HALF PRODUCT COUNT :X: :Y:
20 MARK :X: :Y:
END

←#DOALL
*** 101 *** MARK 1 0
111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111
1
BREAK
I WAS AT LINE 10 IN MARK
←LIST MARK

TO MARK :CHAR: :N:
10 TYPE :CHAR:
20 TEST IS DIFFERENCE :N: 1 0
30 IFTRUE STOP
40 MARK :CHAR: DIFFERENCE :N: 1
END

←.
```



```
TO UPD :N:
10 MIDDLE 50 "X" :N:
20 PRINT ""
30 TEST IS :N: 1
40 IFTRUE STOP
50 MAKE "N" DIFFERENCE :N: 2
60 UPD :N:
END
```

Figure 16.

```
1
              MARK OF "1" AND "-55"
1
MARK OF "1" AND "-56"
1
  MARK OF "1" AND "-57"
1
    MARK OF "1" AND "-58"
1
      MARK OF "1" AND "-59"
1
        MARK OF "1" AND "-60"
1
          MARK OF "1" AND "-61"
1
            MARK OF "1" AND "-62"
1
              MARK OF "1" AND "-63"
1
MARK OF "1" AND "-64"
BREAK

←*DOALL
*** 110 *** ERASE ALL TRACES
*** 111 *** UPD 5
                       XXXXX
                        XXX
                         X
*** 112 *** TO DIAMOND
*** 113 *** 10 NUM 8
*** 114 *** 20 UPD 15
*** 115 *** END
```



```
TO DIAMOND
10 NUM 8
20 UPD 15
END
```

Figure 17.

```
*** 116 *** DIAMOND
                         X
                        XXX
                       XXXXX
                      XXXXXXX
                     XXXXXXXXX
                    XXXXXXXXXXX
                   XXXXXXXXXXXXX
                  XXXXXXXXXXXXXXX
                  XXXXXXXXXXXXXXX
                   XXXXXXXXXXXXX
                    XXXXXXXXXXX
                     XXXXXXXXX
                      XXXXXXX
                       XXXXX
                        XXX
                         X
*** 117 *** EDIT DIAMOND
*** 118 *** 10 NUM 9
*** 119 *** END
*** 120 *** DIAMOND
            .  .  .
            .  .  .
            .  .  .


←*DOALL
*** 121 *** ERASE DIAMOND
*** 122 *** TO DIAMOND : L : : V :
*** 123 *** 10 NUM : L :
*** 124 *** 20 UPD : V :
*** 125 *** END
```

Figure 18.

```
*** 126 *** DIAMOND 9 15
                         X
                        XXX
                       XXXXX
                      XXXXXXX
                     XXXXXXXXX
                    XXXXXXXXXXX
                   XXXXXXXXXXXXX
                  XXXXXXXXXXXXXXX
                 XXXXXXXXXXXXXXXXX
                  XXXXXXXXXXXXXXX
                   XXXXXXXXXXXXX
                    XXXXXXXXXXX
                     XXXXXXXXX
                      XXXXXXX
                       XXXXX
                        XXX
                         X

←#DOALL
*** 128 *** EDIT DIAMOND
*** 129 *** END
*** 130 *** ERASE DIAMOND
*** 131 *** TO DIAMOND : L :
*** 132 *** 10 NUM : L :
*** 133 *** 20 UPD SUM : L : DIFF : L : 1
*** 134 *** END
*** 135 *** DIAMOND 5
                         X
                        XXX
                       XXXXX
                      XXXXXXX
                     XXXXXXXXX
                     XXXXXXXXX
                      XXXXXXX
                       XXXXX
                        XXX
                         X
```

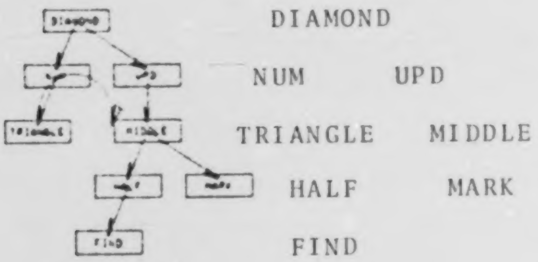
Figure 19.

```
*** 136 *** EDIT DIAMOND
*** 137 *** 20 UPD SUM : L : DIFF : L : 3
*** 138 *** END
*** 139 *** DIAMOND 3
                         X
                        XXX
                       XXXXX
                        XXX
                         X
*** 140 *** DIAMOND 7
                         X
                        XXX
                       XXXXX
                      XXXXXXX
BREAK

             :  :  :
```

```
*** 151 *** DIAMOND RANDOM
                         X
                        XXX
                       XXXXX
                        XXX
                         X
*** 152 *** LIST MIDDLE

TO MIDDLE :N: :X: :Y:
10 MARK "■" DIFFERENCE HALF :N: HALF PRODUCT COUNT :X: :Y:
20 MARK :X: :Y:
END

*** 153 *** LIST TRIANGLE

TO TRIANGLE :N:
10 MIDDLE 50 "X" :NU:
20 PRINT ""
30 TEST IS SUM :N: DIFFERENCE :N: 1 :NU:
40 IFTRUE STOP
50 MAKE "NU" SUM :NU: 2
END

*** 154 *** P TIME
8:38 AM
*** 155 *** LOGOUT
```

Figure 20.

```
←$WHERE
AT DRIBBLE LINE 155
     WHICH IS
LOGOUT
THE PROCEDURES MARK SUPERMARK RECTANGLE SUBRECTANGLE FIND HALF DELETE
STRIPE MIDDLE NUM TRIANGLE UPD DIAMOND DIAMOND DIAMOND HAVE BEEN DEFINED
←$DISPLAY "DIAMOND"
←
```


DIAMOND
NUM
UPD
TRIANGLE
MIDDLE
HALF
MARK
FIND

```
←$DOALL
*** 158 ***
****END-OF-FILE****
←
```

Figure 21.

4. Example 3

This dribble file documents the development of a set of programs written to generate grammatic sentences with randomly chosen constituents. Figure 22 shows the execution of most of the lines of the dribble file. Specifically, the procedures CHOOSE, RANDOMCHOOSE, GETNOUN, GETVERB, GETADJ, GETCON, MAKEDICTIONARY, LITTLESENTENCE, SIMPLESENTENCE, BIGSENTENCE, TALK, KEEPTALKING, and TALKALOT have been defined. Most of these procedures are short (1 - 4 lines) and their effects are quite transparent.

CHOOSE has two inputs -- an index and a list of words. It selects the word in the list specified by the index. RANDOMCHOOSE has a list of words as input. It uses CHOOSE with a random digit as an index, to select a word in a ten-word list at random. GETNOUN, GETVERB, GETADJ, and GETCON are procedures that use RANDOMCHOOSE to randomly select a noun, verb, adjective, or connective, respectively, from correponding word lists of such constituents. MAKEDICTIONARY creates these four word lists. LITTLESENTENCE generates a random sentence of the form <noun> <verb>, using GETNOUN and GETVERB.

SIMPLESENTENCE uses GETADJ and LITTLESENTENCE to generate a random sentence of the form <adjective><noun><verb>. BIGSENTENCE :SIZE: generates a random sentence of the specified number of clauses, each of which is generated by LITTLESENTENCE. The clauses are joined by connectives randomly generated by GETCON. The procedure TALK simply invokes MAKEDICTIONARY and then BIGSENTENCE. KEEPTALKING calls TALK (with a random digit for :SIZE:) and then calls itself. Its effect is to generate randomly composed sentences of random size indefinitely.

TALKALOT :N: calls TALK (with a RANDOM input) :N: times; thus it generates :N: random sentences of random size.

Figure 23 shows student trials of more of these procedures, following the execution of a $WHERE by the analyst. The student tries TALK with :SIZE: 1, then 3, then 2; then he tries TALKALOT with :N:=2; then he tries KEEPTALKING, whose execution is interrupted with a BREAK at the end of the figure.

In Figure 24 the analyst diagrams the procedure structures for LITTLESENTENCE and SIMPLESENTENCE. In Figure 25 he displays the diagram of one of the top level procedures, KEEPTALKING. The conventions used in these diagrams are straightforward. Each box is associated with the procedure whose name labels the box. A directed line from box $\boxed{P}$ to box $\boxed{Q}$ denotes that procedure P uses procedure Q. When P uses Q and both are in the same row (level) of the diagram (because both are used by some other procedure) then a directed arc is used to join $\boxed{P}$ and $\boxed{Q}$.

The usefulness of these diagrams in giving the analyst a global overview of the program structure is evident here. Most systems of programs, both those generated in extended student projects and those produced in professional programming work, are a great deal more complex than the sentence generator of this example. Typically there are larger numbers of programs, program interconnections, levels of call, and recursive parts; and the components are larger and more opaque. Diagrams are especially useful when used to aid the analysis of these more complex program structures.

```
TO CHOOSE : INDEX : : LIST :
10 IF : INDEX : = 0 OUTPUT FIRST OF : LIST : ELSE OUTPUT
CHOOSE ( : INDEX : - 1 ) ( BUTFIRST OF : LIST : )
END
TO RANDOMCHOOSE : LIST :
10 OUTPUT CHOOSE OF RANDOM AND : LIST :
END
TO GETNOUN
1 OUTPUT RANDOMCHOOSE : NOUNS :
END
TO GETVERB
1 OUTPUT RANDOMCHOOSE : VERBS :
END
TO GETADJ
1 OUTPUT RANDOMCHOOSE : ADJECTIVES :
END
TO GETCON
1 OUTPUT RANDOMCHOOSE : CONNECTIVES :
END
TO MAKEDICTIONARY
1 MAKE " NOUNS " " PROGRAMS BUGS CHILDREN GURUS TURTLES
LOVERS TRUTHS POEMS GUGGLES STARS "
2 MAKE " VERBS " " WORK HURT RETURN LEARN PLAY SING FALL
OUTPUT CELEBRATE LAUGH "
3 MAKE " ADJECTIVES " " INCREDIBLE GOOD NOISY BEAUTIFUL OLD
OVOID TRUE ABSTRACT OBVIOUS GRONCHY "
4 MAKE " CONNECTIVES " " SINCE AND WHILE THOUGH BUT AS YET IF
UNTIL BECAUSE "
END
TO LITTLESENTENCE
10 OUTPUT SENTENCE OF GETNOUN AND GETVERB
END
TO SIMPLESENTENCE
10 OUTPUT SENTENCE OF GETADJ AND LITTLESENTENCE
END
TO BIGSENTENCE : SIZE :
10 IF : SIZE : = 1 OUTPUT SIMPLESENTENCE ELSE OUTPUT
SENTENCES SIMPLESENTENCE GETCON BIGSENTENCE ( : SIZE : - 1 )
END
TO TALK : SIZE :
1 MAKEDICTIONARY
2 PRINT BIGSENTENCE : SIZE :
END
TO KEEPTALKING
1 TALK RANDOM
2 KEEPTALKING
END
TO TALKALOT : N :
10 IF : N : = 0 STOP ELSE TALK RANDOM
20 TALKALOT : N : - 1
END
```

Figure 22.

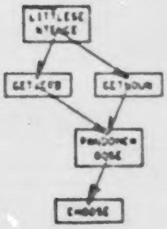
```
←#WHERE
AT DRIBBLE LINE 46
     WHICH IS
END
THE PROCEDURES CHOOSE RANDOMCHOOSE GETNOUN GETVERB GETADJ GETCON
MAKEDICTIONARY LITTLESENTENCE SIMPLESENTENCE BIGSENTENCE TALK
KEEPTALKING TALKALOT HAVE BEEN DEFINED

←#DOTO 75
*** 49 *** TALK 1
BEAUTIFUL STARS CELEBRATE
*** 50 *** TALK 3
NOISY CHILDREN WORK WHILE GRONCHY CHILDREN OUTPUT AS TRUE TRUTHS
CELEBRATE
*** 51 *** TALK 2
TRUE BUGS CELEBRATE BECAUSE GRONCHY GURUS RETURN
*** 52 *** TALKALOT 2
OVOID GURUS SING IF BEAUTIFUL GUGGLES SING WHILE GOOD POEMS RETURN SINCE
OLD TURTLES HURT THOUGH NOISY LOVERS WORK BECAUSE GRONCHY GURUS LEARN
SINCE OLD PROGRAMS FALL BECAUSE OLD GUGGLES FALL
BEAUTIFUL LOVERS SING IF NOISY GUGGLES CELEBRATE AS INCREDIBLE PROGRAMS
RETURN AND INCREDIBLE POEMS FALL YET BEAUTIFUL GURUS FALL IF GOOD BUGS
SING THOUGH BEAUTIFUL PROGRAMS RETURN
*** 53 *** KEEPTALKING
ABSTRACT BUGS SING UNTIL TRUE TURTLES OUTPUT UNTIL NOISY POEMS LEARN
BREAK
I WAS AT LINE 10 IN CHOOSE
←
```

```
TO TALKALOT :N:
10 IF :N:=0 STOP ELSE TALK RANDOM
20 TALKALOT :N:-1
END
```

```
TO TALKALOT :N:
10 IF :N:=0 STOP ELSE TALK RANDOM
20 TALKALOT :N:  1
END
```

Figure 23.

```
*@DISPLAY "LITTLESENTENCE"
*
```


LITTLESENTENCE
GETVERB
GETNOUN
RANDOMCHOOSE
CHOOSE

```
*@DISPLAY "SIMPLESENTENCE"
*
```

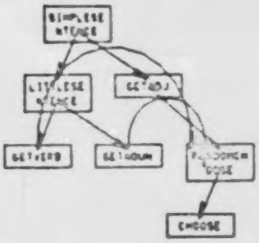
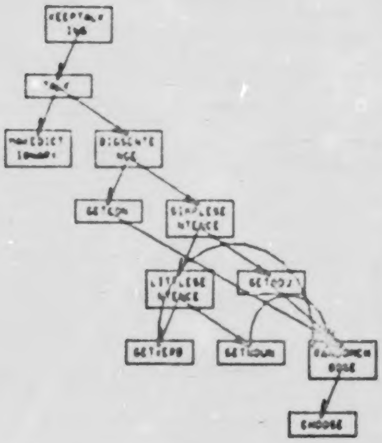

SIMPLESENTENCE
LITTLESENTENCE
GETADJ
GETVERB
GETNOUN
RANDOMCHOOSE
CHOOSE

Figure 24.

```
•1DISPLAY "KEEPTALKING"
•
 •
```



KEEPTALKING

TALK

MAKEDICTIONARY BIGSENTENCE

GETCON SIMPLESENTENCE

LITTLESENTENCE GETADJ

GETVERB GETNOUN RANDOMCHOOSE

CHOOSE

Figure 25.

5. Example 4

The system of programs in this dribble file deals with the extrapolation of sequence. Given a number sequence such as 1 2 3 4, the system tries to "guess" the next term so as to satisfy the user. It does this by trying out various extrapolattion procedures to see if any of them successfully extrapolate the known terms, i.e., extrapolate 1 to 2, 2 to 3, and 3 to 4 in the example given. If not, the system asks the user for the extrapolation procedure he was using and, after assuring that it successfully extrapolates the known terms, adds it to its growing repertory of trial procedures.

These programs are developed in the first part of the dribble file (Figures 26 and 27). The system comprises seven programs: CASSANDRA, EXECUTE, CHECK, SECOND, SCAN, SUCCESS, and CONTINUE. CASSANDRA (which is defined in lines ***2*** through ***9***) is the top level program. It starts by asking its user to type in a sequence. After accepting the sequence, it uses the procedure SCAN to search its existing extrapolation procedures (in :PROCEDURE BANK:) to see if any of these successfully extrapoiate the given sequence. If so, it stops; otherwise, it asks the user to define the correct extrapolation procedure and, when done, to resume operation by calling the procedure CONTINUE.

EXECUTE and SECOND (lines ***10*** through ***12*** and ***20*** through ***22***) are small utility procedures. EXECUTE :PROCEDURE: :INPUT: performs the specified (single input) :PROCEDURE: with the specified :INPUT: and outputs the result. SECOND :LIST: outputs the second element in the specified :LIST:.

CHECK :PROCEDURE: :SEQUENCE: (Lines ***13*** through ***19***) tests whether the specified procedure successfully extrapolates the successive terms in the specified sequence. If so, it outputs "TRUE"; if not, "FALSE". For example, consider the procedure ADD1 whose output is 1 plus its input:

```
TO ADD1 :INPUT:
1 OUTPUT (:INPUT: + 1)
END
```

CHECK of ADD1 "11 12 13" outputs "TRUE" (since ADD1 of 11 = 12 and ADD1 of 12 = 13) but CHECK of ADD1 "2 4 6" outputs "FALSE" (since ADD1 of 2 ≠ 4).

SCAN :SEQUENCE: :PROCEDURES: (lines ***24*** through ***33***) scans through the given list of extrapolation procedures one at a time with CHECK, to find whether one of these procedures successfully extrapolates the known terms in the given sequence. If so, it uses this procedure to extrapolate a next term and (using the procedure SUCCESS) determines whether or not the extrapolated next term is acceptable to the user. If unacceptable, it tries again with the remaining procedures.

The procedure SUCCESS (Figure 27) simply asks the user whether the extrapolated next term is acceptable and outputs "TRUE" or "FALSE" accordingly. The last procedure in the set, CONTINUE, is called by the user after he inputs a new extrapolation procedure. CONTINUE asks for the name of the new procedure, uses CHECK to check its correctness on the given user sequence. If it is correct, it is added to the list of extrapolation procedures. At this point CONTINUE, as the most recently defined procedure, is displayed in mini-print at the lower left of the display.

Figure 28 shows the beginning of the student's execution of these procedures. The analyst begins reviewing the student's work with a $DOALL. Immediately a difficulty appears. The execution of line ***53*** (which calls CASSANDRA to start the system) requires the user of the system to provide an input. CASSANDRA types SEQUENCE?... and waits for the user to type in a sequence. What is the analyst to do at this point? He has two options -- he can provide his own input or he can replicate the input used by the student in the next line of the dribble file. The analyst probably wants to see the student's input, even if he decides to use a different one.

Thus, it is valuable for the analyst to have a facility for looking ahead in the dribble file to see the student's inputs on the next line (or lines if a multi-line input chain is involved). During this look-ahead process, execution should be suspended until the analyst has previewed as many lines as he wishes to see. And he should be able to indicate which of these student input lines he will want to be ignored during the subsequent execution (because, for example, he wishes to replace these with his own inputs).

Following the exemplars of user-defined analysis procedures described in Part 3, the analyst of this example has written such a lookahead procedure, called $LOOKAHEAD. The procedure is shown and discussed later; first we illustrate its use in the subsequent analysis. To suspend operation of the dribble file execution the analyst types BREAK (on line 4 of Figure 28). LOGO responds (I WAS AT LINE 20 IN CASSANDRA) and the analyst invokes the lookahead procedure by typing $LOOKAHEAD.

The procedure asks the analyst if he wishes to look ahead to the next line (MORE?...). He responds with YES. The procedure then types dribble file line ***54*** (prefixed with [[[[[to indicate that it is a previewed line). This line is the student's input sequence 1 2 3 4. The procedure then asks whether this line is to be ignored during subsequent execution (IGNORE?...). The analyst responds YES. He is then asked whether he wants to look ahead to the next line (MORE?...). He responds YES and is shown line ***55*** (which is the title line of the procedure ADD1, an extrapolation procedure being defined by the student). The analyst indicates that he does not wish to ignore this line. Then he responds NO to MORE?..., since he does not wish to look ahead any more at this point. $LOOKAHEAD then types **RESUME AT ***53*** and continues processing there, at the point where it had left off before.

In response to SEQUENCE?... the analyst types 1 2 3 4. The program then asks for the definition of a procedure that generates that sequence (I CAN'T DO THAT ONE ...). The analyst invokes the next several lines of the dribble file, with a $DOALL. This enters the student's extrapolation procedure ADD1, and then invokes (line ***59***) the procedure CONTINUE, which asks its user for an input (WHAT IS THE NAME OF YOUR NEW PROCEDURE?).

At this point the analyst types BREAK to interrupt processing and calls $LOOKAHEAD to preview the next two lines in the file. These lines enter the name of the student's extrapolation procedure (ADD1) and ask for the list of student extrapolation procedures (PRINT :PROCEDURE BANK:) . The analyst then resumes execution with a $DOALL, and executes the dribble file through line ***62*** which calls CASSANDRA. After CASSANDRA's request for the input of a sequence, the analyst once more BREAK's (end of Figure 28).

As shown in Figures 29 and 30, he carries on the analysis of the student's work in this fashion using $LOOKAHEAD extensively, throughout the rest of the dribble file.

Figure 29 shows the student's trial of CASSANDRA with the sequences 1117 1118 1119 and 1 11 111 1111 and his definition of another extrapolation procedure, TAG1. Figure 30 shows a later portion of the dribble file, by which time the extrapolation procedures ADD1, TAG1, TAGLAST, and TIMES2 have been introduced (line ***95***). The last few lines show the student input of the sequence 1 2, his rejection of the extrapolated next term 3 (which was generated by the extrapolation procedure ADD1), and his acceptance of the extrapolated next term 4 (generated by the procedure TIMES2).

The procedure $LOOKAHEAD, and its main subprocedures are listed in Figure 31. These are very transparent procedures. $LOOKAHEAD1 :N: processes the :N:th line ahead of the current one. If this line is not to be previewed (a negative response to MORE?...), the lookahead process is terminated, and execution is resumed at the point where lookahead started (after spacing two blank lines with $SKIP 2 to set off the interrupted execution lines). If this line is to be previewed, it is displayed by $DISP. If it is to be ignored during subsequent execution, the comment "IGNORE" is appended (using $ADD) to the contents of "(dribble no) D" (which denotes a fatal error, as described in the section of the User's Guide dealing with parsing).

Figures 32 and 33 show the last part of the analysis. At the top of Figure 32 the analyst executes a $WHERE to list all the procedures defined at the end of the student's session. The analyst then invokes $ALLDESCR to list all the descriptors used in the dribble file lines -- these have previously been entered in (dribble no) "B". The system responds with the single descriptor RECURSIVE. The analyst then executes $FINDLINES 1 "RECURSIVE" to list all lines, starting with line 1, which contain that descriptor. The system prints out the associated lines -- 18, 27, 32, and 56. The analyst next prints "RECURSIVE LIST: to list all recursive procedures. These are CHECK, SCAN, and ADD1.

It is interesting to note that ADD1 is listed as recursive because of a lack of sophistication in the procedure $EXAMINEEL that generates the RECURSIVE descriptor. In dribble file line ***56*** the ADD1 title line was changed from TO ADD1 to TO ADD1 :N:. $EXAMINEEL simply sees the repeat of the procedure name ADD1 in this line, failing to observe that this line supplants the original title line.

The blank procedure forms $EXAMINEEL are filled in by the analyst to test whether a line is recursing and, if so, to label it with the descriptor "RECURSIVE". The blank procedure form $NICEP is filled in by the analyst to search (dribble no) "B" descriptor lines for the descriptor "RECURSIVE" and to enter the names of procedures having such lines on "RECURSIVE LIST:. These procedures, along with the blank procedure form $NOTE, are listed in Figure 32.

$NOTE is filled in by the analyst to modify the display names of recursive procedures by prefixing and suffixing them with **. With this modification, procedure structure diagrams explicitly label their recursive components. In Figure 33 the analyst has displayed the structures CASSANDRA and CONTINUE. These diagrams clearly show that the procedures SCAN and CHECK are recursive.

```
*** 2 *** TO CASSANDRA
*** 3 *** 10 TYPE " SEQUENCE?... "
*** 4 *** 20 MAKE " SEQUENCE " REQUEST
*** 5 *** 30 PRINT : EMPTY :
*** 6 *** 40 TEST SCAN OF : SEQUENCE : AND : PROCEDURE BANK :
*** 7 *** 50 IFTRUE STOP
*** 8 *** 60 PRINT " I CAN'T DO THAT ONE. TELL ME HOW TO DO IT ( BY
DEFINING A PROCEDURE FOR GENERATING THE SEQUENCE ) . WHEN YOU'RE DONE
PLEASE TYPE 'CONTINUE'. "
*** 9 *** END
*** 10 *** TO EXECUTE : PROCEDURE : AND : INPUT :
*** 11 *** 10 DO SENTENCE OF " OUTPUT " AND ( SENTENCE OF : PROCEDURE :
AND : INPUT : )
*** 12 *** END
*** 13 *** TO CHECK : PROCEDURE : AND : SEQUENCE :
*** 14 *** 10 TEST EMPTYP OF BUTFIRST OF : SEQUENCE :
*** 15 *** 20 IFTRUE OUTPUT " TRUE "
*** 16 *** 30 TEST IS ( SECOND OF : SEQUENCE : ) EXECUTE OF : PROCEDURE
: AND ( FIRST OF : SEQUENCE : )
*** 17 *** 40 IFFALSE OUTPUT " FALSE "
*** 18 *** 50 OUTPUT CHECK OF : PROCEDURE : AND ( BUTFIRST OF : SEQUENCE
: )
*** 19 *** END
*** 20 *** TO SECOND : LIST :
*** 21 *** 10 OUTPUT FIRST OF BUTFIRST OF : LIST :
*** 22 *** END
*** 23 *** TO SCAN : SEQUENCE : AND : PROCEDURES :
*** 24 *** 10 TEST EMPTYP OF : PROCEDURES :
*** 25 *** 20 IFTRUE OUTPUT " FALSE "
*** 26 *** 30 TEST CHECK OF ( FIRST OF : PROCEDURES : ) AND : SEQUENCE :
*** 27 *** 40 IFFALSE OUTPUT SCAN OF : SEQUENCE : AND ( BUTFIRST OF :
PROCEDURES : )
*** 28 *** 50 MAKE " NEXT TERM " EXECUTE OF ( FIRST OF : PROCEDURES : )
AND ( LAST OF : SEQUENCE : )
*** 29 *** 60 TEST SUCCESS OF : NEXT TERM :
*** 30 *** 70 IFTRUE PRINT SENTENCE " THE WINNING PROCEDURE HAS " (
FIRST OF : PROCEDURES : )
*** 31 *** 80 IFTRUE OUTPUT " TRUE "
*** 32 *** 90 OUTPUT SCAN OF : SEQUENCE : AND ( BUTFIRST OF : PROCEDURES
: )
*** 33 *** END
```

Figure 26.

```
*** 34 *** TO SUCCESS : TERM :
*** 35 *** 10 TYPE SENTENCES " IS THE NEXT TERM " : TERM : " ? "
*** 36 *** 20 MAKE " ANSWER " REQUEST
*** 37 *** 30 TEST IS : ANSWER : " YES "
*** 38 *** 40 IFTRUE OUTPUT " TRUE "
*** 39 *** 50 OUTPUT " FALSE "
*** 40 *** END
*** 41 *** TO CONTINUE
*** 42 *** 10 TYPE " WHAT IS THE NAME OF YOUR NEW PROCEDURE? "
*** 43 *** 20 MAKE " NEW PROCEDURE " REQUEST
*** 44 *** 30 PRINT : EMPTY :
*** 45 *** 40 TEST CHECK OF : NEW PROCEDURE : AND : SEQUENCE :
*** 46 *** 50 IFFALSE PRINT SENTENCES " NO, " : NEW PROCEDURE : " DOES
NOT GENERATE THE SEQUENCE THAT YOU GAVE ME. TRY YOUR PROCEDURE ON
SUCCESSIVE TERMS AND YOU WILL SEE THAT IT DOESN'T WORK. "
*** 47 *** 60 IFTRUE MAKE " PROCEDURE BANK " SENTENCE OF : PROCEDURE
BANK : AND : NEW PROCEDURE :
*** 48 *** 70 IFTRUE PRINT " THANKS FOR THE NEW RULE. "
@*DOLINE
*** 49 *** END
*.
```

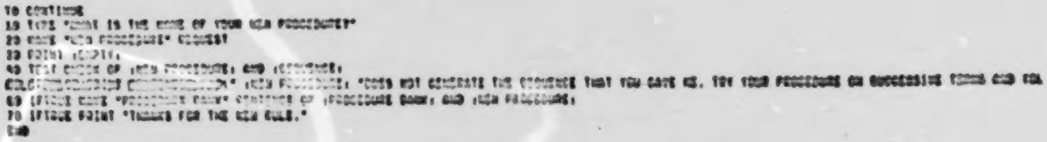


Figure 27.

```
←#DOALL
*** 53 *** CASSANDRA
SEQUENCE?...
BREAK
I WAS AT LINE 20 IN CASSANDRA
←#LOOKAHEAD
MORE?...YES
[[[[[*** 54 *** 1 2 3 4
IGNORE?...YES
MORE?...YES
[[[[[*** 55 *** TO ADD1
IGNORE?..:NO
MORE?...NO
**RESUME AT*** 53 ***


SEQUENCE?...1 2 3 4

I CAN'T DO THAT ONE. TELL ME HOW TO DO IT ( BY DEFINING A PROCEDURE FOR
GENERATING THE SEQUENCE ) . WHEN YOU'RE DONE PLEASE TYPE 'CONTINUE'.
←#DOALL
*** 55 *** TO ADD1
*** 56 *** TITLE TO ADD1 : N :
*** 57 *** 1 OUTPUT : N : + 1
*** 58 *** END
*** 59 *** CONTINUE
WHAT IS THE NAME OF YOUR NEW PROCEDURE?
BREAK
I WAS AT LINE 20 IN CONTINUE
←#LOOKAHEAD
MORE?...Y
[[[[[*** 60 *** ADD1
IGNORE?...Y
MORE?...Y
[[[[[*** 61 *** PRINT : PROCEDURE BANK :
IGNORE?...N
MORE?...N
**RESUME AT*** 59 ***


WHAT IS THE NAME OF YOUR NEW PROCEDURE?ADD1

THANKS FOR THE NEW RULE.
←#DOALL
*** 61 *** PRINT : PROCEDURE BANK :
ADD1
*** 62 *** CASSANDRA
SEQUENCE?...
BREAK
I WAS AT LINE 20 IN CASSANDRA
```

Figure 28.

```
$←LOOKAHEAD
MORE?...Y
[[[[[*** 63 *** 1117 1118 1119
IGNORE?...Y
MORE?...Y
[[[[[*** 64 *** YES
IGNORE?...Y
MORE?...Y
[[[[[*** 65 *** CASSANDRA
IGNORE?...N
MORE?...N
**RESUME AT*** 62 ***


SEQUENCE?...1 2 3 4.

IS THE NEXT TERM 5 ?YES
THE WINNING PROCEDURE WAS ADD1
←$DOALL
*** 65 *** CASSANDRA
SEQUENCE?...
BREAK
I WAS AT LINE 20 IN CASSANDRA
←$LOOKAHEAD
MORE?...Y
[[[[[*** 66 *** 1 11 111 1111
IGNORE?...Y
MORE?...Y
[[[[[*** 67 *** TO TAG1 : N :
IGNORE?...N
MORE?...N
**RESUME AT*** 65 ***


SEQUENCE?...1 11 111 1111

I CAN'T DO THAT ONE. TELL ME HOW TO DO IT ( BY DEFINING A PROCEDURE FOR
GENERATING THE SEQUENCE ) . WHEN YOU'RE DONE PLEASE TYPE 'CONTINUE'.
←$DOALL
*** 67 *** TO TAG1 : N :
*** 68 *** 1 OUTPUT WORD OF : N : AND 1
*** 69 *** END
*** 70 *** CONTINUE
WHAT IS THE NAME OF YOUR NEW PROCEDURE?.
```

Figure 29.

```
←$DOALL
*** 95 *** PRINT : PROCEDURE BANK :
ADD1 TAG1 TAGLAST TIMES2
*** 96 *** CASSANDRA
SEQUENCE?...
BREAK
I WAS AT LINE 20 IN CASSANDRA
←$LOOKAHEAD
 :  :  :

**RESUME AT*** 96 ***


SEQUENCE?...99 198 396

IS THE NEXT TERM 792 ?YES
THE WINNING PROCEDURE WAS TIMES2
←$DOALL
*** 99 *** CASSANDRA
SEQUENCE?...
BREAK
I WAS AT LINE 20 IN CASSANDRA
$L←OOKAHEAD
MORE?...Y
[[[[[*** 100 *** 1 2
IGNORE?...Y
MORE?...Y
[[[[[*** 101 *** NO
IGNORE?...Y
MORE?...Y
[[[[[*** 102 *** YES
IGNORE?...Y
MORE?...Y
[[[[[*** 103 *** SAVE CASS2 CASS2
IGNORE?...N
MORE?...N
**RESUME AT*** 99 ***


SEQUENCE?...1 2

IS THE NEXT TERM 3 ?NO
IS THE NEXT TERM 4 ?YES
THE WINNING PROCEDURE WAS TIMES2
←$DOALL
*** 104 *** $ERASEALL
*** 105 ***
****END-OF-FILE****
←.
```

Figure 30.

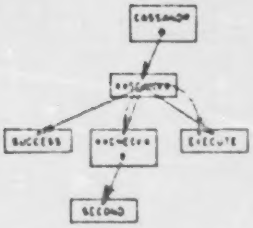
```
←LIST #LOOKAHEAD

TO #LOOKAHEAD
10 #LOOKAHEAD1 1
END

←LIST #LOOKAHEAD1

TO #LOOKAHEAD1 :N:
10 TYPE "MORE?..."
20 TEST #YESP REQUEST
30 IFFALSE PRINT SENTENCES "**RESUME AT***" :DRIBBLE NO: "***"
40 IFFALSE #SKIP 2
50 IFFALSE DO THING SENTENCE :DRIBBLE NO: "N"
60 IFFALSE STOP
70 TYPE "[[[[["
80 #DISP SUM :DRIBBLE NO: :N:
90 TYPE "IGNORE?..."
100 IF #YESP REQUEST #ADD SENTENCE SUM :DRIBBLE NO: :N: "D" "IGNORE"
110 #LOOKAHEAD1 SUM :N: 1
END

←LIST #YESP

TO #YESP :ANS:
10 IF IS :ANS: "Y" OUTPUT "TRUE"
20 IF IS :ANS: "YES" OUTPUT "TRUE"
30 OUTPUT "FALSE"
END

←LIST #ADD

TO #ADD :PLACE: :MES:
10 MAKE :PLACE: SENTENCE THING :PLACE: :MES:
END
```

Figure 31.

```
←#WHERE
AT DRIBBLE LINE 105
     WHICH IS

THE PROCEDURES CASSANDRA EXECUTE CHECK SECOND SCAN SUCCESS CONTINUE ADD1
TAG1 TAGLAST TIMES2 HAVE BEEN DEFINED
←#ALLDESCR
RECURSIVE
←#FINDLINES 1 "RECURSIVE"
RECURSIVE IN 18 --- 50 OUTPUT CHECK OF : PROCEDURE : AND ( BUTFIRST OF :
SEQUENCE : )
RECURSIVE IN 27 --- 40 IFFALSE OUTPUT SCAN OF : SEQUENCE : AND (
BUTFIRST OF : PROCEDURES : )
RECURSIVE IN 32 --- 90 OUTPUT SCAN OF : SEQUENCE : AND ( BUTFIRST OF :
PROCEDURES : )
RECURSIVE IN 56 --- TITLE TO ADD1 : N :

←PRINT :RECURSIVE LIST:
CHECK SCAN ADD1
←.


←LIST #EXAMINEEL

TO #EXAMINEEL
10 IF NOT IS FIPST :CURRENT LINE: :CURRENT PROC: STOP
20 IF NOT #MP "RECURSIVE" THING SENTENCE :DRIBBLE NO: "B" #ADD SENTENCE
   :DRIBBLE NO: "B" "RECURSIVE"
END

←
←.

←LIST #NOTE

TO #NOTE :PROC:
10 IF #RECURSEP :PROC: OUTPUT WORD WORD "**" :PROC: "**"
20 OUTPUT :PROC:
END

←LIST #NICEP

TO #NICEP
10 IF #MP "RECURSIVE" THING SENTENCE :DRIBBLE NO: "B" MAKE "RECURSIVE
   LIST" #UNION :CURRENT PROCEDURE: :RECURSIVE LIST:
20 OUTPUT "TRUE"
END
```

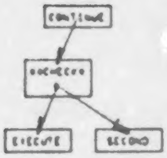
Figure 32.

```
*$DISPLAY "CASSANDRA"
*
```



```
                    CASSANDRA

                      **SCAN**

SUCCESS      **CHECK**        EXECUTE

                SECOND
```

```
*$DISPLAY "CONTINUE"
*
```



```
                CONTINUE

                **CHECK**

EXECUTE           SECOND
```

Figure 33.

APPENDIX A

LOGO REFERENCE MANUAL

1. A Look at LOGO

We introduce LOGO by writing several small procedures. The following examples serve to show what LOGO "looks like". Several features are used without definition or even explanation, where we think their meanings are clear from context. All of LOGO is comprehensively described in later sections.

LOGO, as an interpretive language, can execute single commands directly. Thus,

```
←PRINT SUM OF 2 AND 2          (The user's typing is underlined)
4
```

But, the most important feature of LOGO is that such commands can be incorporated in user-written procedures. The definition of any procedure results in an object which is treated just like any primitive. Thus, in a very real sense, as the user writes his own procedures, he is gradually extending the basic language to more exactly fill his needs.

A very simple (although by no means simplest) procedure, for example, prints the double of its input.

```
TO DOUBLE /N/
10 PRINT SUM OF /N/ AND /N/
END
```

This procedure, DOUBLE, is now "part" of LOGO.

```
←DOUBLE 123
246
←DOUBLE WORD OF 1 AND 1
22
```

If the concatenation of DOUBLE with other procedures is desired, DOUBLE should OUTPUT rather than PRINT its results; OUTPUT meaning that the result is given to the calling procedure. The modified procedure is:

```
TO DOUBLE /N/
10 OUTPUT SUM OF /N/ AND /N/
END
```

This new version of DOUBLE can be used in direct commands

```
←PRINT DOUBLE DOUBLE DOUBLE 3
24
```

or can be used as the basis for other procedures,

```
TO QUADRUPLE /N/
10 OUTPUT DOUBLE OF DOUBLE OF /N/
END
```

and so on. This very natural use of functions in LOGO is particularly valuable, since to program a problem a user can keep on breaking it up until he sees subproblems which he feels will be easy to program. This heuristic is used by sophisticated problem-solvers generally, whether or not computer programming is involved.

An extension of this LOGO facility for using procedures in defining other procedures is its ability to handle recursive procedure definitions. The recursion can be a linear one, which is equivalent to iteration, as in the following procedure which calculates the factorial function: $n! \equiv n \cdot (n-1) \cdots 2 \cdot 1$.

```
TO FACTORIAL /NUMBER/
10 TEST IS /NUMBER/ 1
20 IF TRUE OUTPUT 1                  (1! ≡ 1)
30 OUTPUT PRODUCT OF /NUMBER/ AND
     (FACTORIAL OF DIFFERENCE OF     (n! = n · (n-1)!)
     /NUMBER/ AND 1)
END
```

```
←PRINT FACTORIAL OF 100
9332621544394415268169923885626670049071596826438162146859296389521
7599993229915608941463976156518286253697920827223758251185210916864
000000000000000000000000
```

LOGO makes few distinctions between numbers and more general types of string. Thus, a procedure to reverse a string looks very much like FACTORIAL.

```
TO REVERSE /STRING/
10 TEST IS /STRING/ /EMPTY/          (The reverse of the empty string
20 IF TRUE OUTPUT /EMPTY/            is the empty string.)
30 OUTPUT WORD OF LAST OF /STRING/(The reverse of the nonempty
     AND (REVERSE OF BUTLAST OF      string is the string formed by
     /STRING/)                       following the last character by
END                                  the reverse of the rest of the
                                     string.)
```

("AND", "OF", and parentheses are optional "noise words" for convenience in writing expressions.)

```
←PRINT REVERSE OF "JABBERWOCKY"
YKCOWREBBAJ
```

2. The LOGO Language

2.0 Mechanics

To start using LOGO, a person must establish communication with the computer and specify that he wishes to work with LOGO. The conventions for doing this vary from system to system and are, therefore, outside the bounds of this manual.

To indicate that it is ready for use, LOGO types a back-arrow (←) either immediately or in response to the pressing of the key labeled "RETURN". This means that LOGO is ready to receive an instruction line. After the user types in the desired line, he again presses the RETURN key. This action returns the carriage and gives the command line to LOGO for execution. The LINE FEED key also returns the carriage, but does not cause command execution; thus an arbitrarily long instruction can be entered.

The typing of a line may be aborted at any time by pressing the key labeled "RUB OUT".

The key labeled "BREAK" is used to stop the execution of the current instruction and return control to the user.

The use of LOGO is terminated by typing the command "GOODBYE" or its abbreviation "GB".

Other LOGO system facilities, such as erasing, editing, and filing, are discussed in the section on program manipulation.

2.1 LOGO Objects and the PRINT Command

LOGO contains two kinds of objects, LOGO words and LOGO sentences. A LOGO word is an arbitrary string of printing teletype characters excluding quote marks. A LOGO sentence is an arbitrary string of LOGO words, separated by spaces. If a user types multiple spaces between words of a LOGO sentence, all but one are automatically eliminated.

To simply print a LOGO object on the teletype, the PRINT command is used.

```
←PRINT "Y!!RCHN4LK"                    (a LOGO word)
Y!!RCHN4LK
←PRINT "32425"                         (a LOGO word)
32425
```

```
←PRINT "HI THERE LOGO"          (a LOGO sentence)
HI THERE LOGO
←PRINT ""                       (the "empty" object)
                                (prints out an empty line)
←
```

LOGO objects are delimited by quote marks. If a word is not so delimited, it is taken as a LOGO command or operation. Integer objects are an exception -- for these, quotes are optional.

```
←PRINT 123450
123450
```

2.2 Constructing a Procedure

Using the "built-in" LOGO commands and operations, such as PRINT, we construct procedures either to manipulate LOGO objects or to produce some desired external effects. For example, a LOGO procedure for drawing a small triangle out of + marks is:

```
←TO TRIANGLE
>5 PRINT "+"
>10 PRINT "++"
>32 PRINT "+++"
>40 PRINT "++++"
>END
←
```

To get LOGO to perform this procedure, we merely type the procedure name:

```
←TRIANGLE
+
++
+++
++++
←
```

A procedure definition includes three different kinds of lines. The first line of the definition is called the title line. It begins with the LOGO command TO which indicates to LOGO that we are writing a procedure whose name immediately follows. (TO is not part of the procedure name.) The procedure name must not be a command or operation currently used by LOGO. (This excludes built-in commands like PRINT as well as user-defined procedures in the current workspace.) And, like the LOGO built-in commands and operations, procedure names must not contain quotes or slashes.

After reading the title line, LOGO types back the wedge mark (>). This indicates that it is ready for the next line of the definition. The actions of the procedure are specified by the instruction lines immediately following the title line. Each of these is prefaced by an integer, a *line number*. Following the line number is the instruction itself, such as a LOGO command. The instructions are not executed during this definition phase -- they are merely entered, checked for local syntax errors, and stored away for subsequent execution. Later, when the procedure is to be performed, LOGO will execute these instruction lines in increasing order, by line number, even if they are not set down in such order. The END line indicates the close of the definition.

2.3 Procedures With Inputs

The procedure TRIANGLE always has the same effect. This contrasts with the built-in PRINT command whose effect varies with its input. As well as procedures with no inputs, like TRIANGLE, we can write procedures that have inputs, for example the following one:

```
TO PSYCH /ANYTHING/
10 PRINT "WHY DO YOU SAY"
20 PRINT /ANYTHING/
30 PRINT "?"
END
```

To use PSYCH, we must give it one input:

```
←PSYCH "YOUR NAME IS ELIZA"
WHY DO YOU SAY
YOUR NAME IS ELIZA
?
←

←PSYCH "2+2=4"
WHY DO YOU SAY
2+2=4
?
←
```

By adding a word or sentence enclosed within slashes -- a placeholder or *dummy input* -- to its title line, we denote that PSYCH requires one input. When PSYCH is used, LOGO replaces the dummy input, wherever it occurs in the procedure definition, by the desired input (such as "2+2=4").

We write and use procedures with more than one input along analogous lines. For example:

```
TO REVERSE /FIRST/ /SECOND/ /THIRD/
10 PRINT /THIRD/
20 PRINT /SECOND/
30 PRINT /FIRST/
END
```

REVERSE needs three inputs.

```
←REVERSE "A" "B" "C"
C
B
A

←REVERSE "ABC"
THERE ARE 2 INPUTS MISSING FOR REVERSE
←
```

In the second example, we typed only one input. (When an illegal instruction is attempted, LOGO responds with an appropriate error message. A list of such messages is included in Section 3.5.)

2.4 LOGO Operations

A LOGO *command* (such as PRINT, TO, END, and GOODBYE) always results in some external action. For example, PRINT causes its input to be printed on the teletype. An *operation*, however, *passes on* a LOGO object for further use within LOGO. For example, the LOGO operation SUM requires two inputs (integers), and *outputs* their sum. This new object can either be used as an input for PRINT, if we wish to have it typed, or as an input to a LOGO operation or procedure.

```
←PRINT PRODUCT OF 2 AND 7
14
←PRINT PRODUCT OF 2 AND (SUM OF 3 AND 4)
14
←
```

Here, the output of SUM, 7, is one of the two inputs to the operation PRODUCT which, in turn, passes its output, 14, to PRINT. [OF and AND are completely optional and are used only to improve clarity and readability of expressions for the user. OF may be used after any command, operation, or procedure name. AND may be used between inputs. Balanced parentheses also may be used to improve readability. (See the section following on composition.)]

LOGO includes operations of different kinds, for example, those for concatenation and decomposition given below.

Concatenation

WORD concatenates two given words, its inputs, to form a new word as its output. SENTENCE concatenates its two inputs, either words or sentences, to form a new sentence as its output.

```
←PRINT WORD OF "UP" AND "DOWN"
UPDOWN
←PRINT SENTENCE OF "UP" AND "DOWN"
UP DOWN
←PRINT SENTENCE OF "GO MAN" AND "GO"
GO MAN GO
←PRINT WORD OF "GO MAN" AND "GO"
THE INPUTS TO WORD MUST NOT BE SENTENCES
←
```

Decomposition

FIRST outputs the first character of its input, if the input is a word (or the first word, if the input is a sentence). BUTFIRST outputs everything *but* the *first* of its input (i.e., the second-through-last characters of word inputs, or the second-through-last words of sentence inputs). LAST and BUTLAST are defined similarly.

```
←PRINT FIRST OF "CAT"
C
←PRINT BUTFIRST OF "CAT"
AT
←PRINT LAST OF "CAT"
T
←PRINT BUTLAST OF "CAT"
CA
←PRINT FIRST OF "FEE FIE FOE FUM"
FEE
←PRINT BUTFIRST OF "FEE FIE FOE FUM"
FIE FOE FUM
←PRINT LAST OF "A"
A
←PRINT BUTLAST OF "A"
                                (LOGO prints the empty word)
←
```

The output of FIRST or LAST is always a LOGO word. The output of BUTFIRST or BUTLAST, however, is of the same type (i.e., LOGO word or sentence) as its input. This means that a single word can sometimes be a LOGO sentence. Thus, the output of BUTFIRST OF "THE CAT" is the LOGO one-word sentence "CAT".

```
←PRINT FIRST OF BUTFIRST OF "THE CAT"
CAT
←
```

Arithmetic

The operations SUM, DIFFERENCE, PRODUCT, and QUOTIENT take two inputs, which must be integers, and output the designated arithmetic result. The operation RANDOM has no input and outputs a digit between 0 and 9 generated in a pseudo-random manner.

```
←PRINT SUM OF 2 AND 98
100
←PRINT DIFFERENCE OF 2 AND 98
-96
←PRINT PRODUCT OF 17 AND 5
85
←PRINT QUOTIENT OF 17 AND 5
3                                (the integer quotient)
←PRINT RANDOM
8
←PRINT RANDOM
2
←
```

Other LOGO "built-in" operations are described later.

2.5 Composition

The inputs of LOGO operations do not have to be literal LOGO objects like "CAT" or "39". Instead, as we saw, they may be outputs of other operations. The following examples illustrate how operations can be composed or "chained".

```
←PRINT SENTENCE OF "A B" AND WORD OF "CD" AND "E"
A B CDE
←PRINT LAST OF (FIRST OF "THE CAT")
E
←PRINT DIFFERENCE OF (WORD OF 11 AND 22) AND 122
1000
←
```

Let's consider how LOGO scans and executes these composite instructions by considering the first example above. LOGO scans an instruction line word-by-word from left to right. During the scanning process it identifies each element and notes its type. Thus:

ELEMENT	TYPE
(1) PRINT	Command, needs one input
(2) SENTENCE	Operation, needs two inputs
(3) OF	"Noise" word
(4) "A B"	LOGO sentence literal
(5) AND	Noise word
(6) WORD	Operation, needs two inputs
(7) OF	Noise word
(8) "CD"	LOGO word literal
(9) AND	Noise word
(10) "E"	LOGO word literal

After the input line is scanned, LOGO returns to the beginning of the line and *executes* it as follows:

(1) Fetch the first element PRINT. Its one input remains to be found: fetch the next element.

(2) PRINT's one input will be the output of SENTENCE. SENTENCE, in turn, needs two inputs: fetch the next element.

(3) OF is a noise word, legal in this context. Ignore it. We are still looking for two inputs.

(4) "A B" is an acceptable first input for SENTENCE. The second input needs to be found for SENTENCE: fetch the next element.

(5) AND is a noise word, legal in this context. Ignore it. One input remains to be found for SENTENCE.

(6) SENTENCE's second input will be the output of WORD. WORD, in turn, needs two inputs -- LOGO words: fetch the next element.

(7) OF is a noise word, legal here. Ignore it. We are still looking for two inputs for WORD.

(8) "CD" is an acceptable first input for WORD: fetch the next element.

(9) AND is a noise word, legal here. Ignore it. We are still looking for the second input for WORD.

(10) "E" is an acceptable second input for WORD. There are no further inputs being sought.

(11) WORD outputs "CDE" to SENTENCE as its second input.

(12) SENTENCE outputs "A B CDE" to PRINT as its one input.

(13) PRINT causes A B CDE to be printed on the teletype page.

(14) The processing of the line is complete. LOGO returns control to the user.

This process can be regarded perhaps more conveniently as a successive simplification of the original line as the outputs of various operations are determined. As an illustration, consider the following example:

```
←PRINT PRODUCT OF SUM OF 2 AND 3 AND DIFFERENCE OF 4 AND 5 AND 6
```

Unlike before, we immediately remove all noise words, thus obtaining:

```
←PRINT PRODUCT SUM 2 3 DIFFERENCE 4 5 6
```

We again go word-by-word from left to right. Finding upon reaching the 3 that we can complete the execution of SUM, we do so, obtaining:

```
←PRINT PRODUCT 5 DIFFERENCE 4 5 6
```

This execution of SUM does not permit any previously encountered operations to be executed so we keep on going to the right. We find that we can next complete the execution of DIFFERENCE. Doing this, we obtain:

```
←PRINT PRODUCT 5 -1 6
```

Again, looking back at previously encountered operations, we find that PRODUCT can now be executed. This results in:

```
←PRINT -5 6
```

Looking back once more, we find we can execute PRINT. We do so, thereby obtaining the printout -5. Since the first command on the original line has been executed, we deem the execution of the line to be complete. There is, however, something remaining on the line and this causes LOGO to print the error message:

```
"6" IS EXTRA
```

LOGO then returns control to the user.

Parentheses

Parentheses may be used in instructions to set off any expression that is, or can be used as, a LOGO input. The following examples of such use are valid:

```
←PRINT SUM (2) 3
←PRINT SUM 2 (3)
←PRINT (SUM 2 3)
←(PRINT SUM 2 3)
```

All these instructions cause the printout 5.

The following examples show some invalid uses of parentheses.

```
←PRINT (SUM) 2 3
←PRINT (SUM 2) 3
←PRINT SUM (2 3)
```

None of the above parenthesized expressions are meaningful LOGO inputs. The attempted execution of these lines will cause LOGO to print error messages such as MATCHING (? and MISSING)?.

Two infix expressions such as

2 + (3 X 4) and (2 + 3) X 4

whose operations and inputs occur in the same order, but which are differently parenthesized, yield different results. Thus, the use of parentheses can change the order of evaluation of infix expressions. This is not so with LOGO prefix expressions. So long as parentheses are used correctly -- that is, to enclose inputs or possible inputs -- they cannot change the result of evaluating any LOGO prefix expression.

2.6 Procedures With Outputs

LOGO operations, by definition, output when they are executed. LOGO procedures can be written which output in just the same way as operations. To do this, the one-input LOGO command OUTPUT is used. Its use is illustrated:

```
TO DUBBLE /SOMETHING/
10 OUTPUT WORD OF /SOMETHING/ AND /SOMETHING/
END

←PRINT DUBBLE "TROUBLE"
TROUBLETROUBLE
```

Note that, as in the case of built-in operations like FIRST or SUM, we need to preface DUBBLE with PRINT to cause the output of DUBBLE to be typed. In line 10 of DUBBLE, OUTPUT causes the result of execution of its line to be passed back as an input to the PRINT command which called DUBBLE. The LOGO command OUTPUT takes a single input and passes this back as an input to the command, operation, or procedure which called the present procedure. OUTPUT then causes resumption of the execution of the line to which it returned its input. At this point the outputting procedure effectively vanishes. Clearly, the command OUTPUT can only be used within a procedure.

User-defined procedures which output can be composed in the same way as built-in operations. Thus:

```
←PRINT SUM OF DUBBLE 3 AND 4
37
←PRINT DUBBLE DUBBLE DUBBLE "B"
BBBBBBBB

←TO QUADRUBBLE /STUFF/
>10 OUTPUT DUBBLE OF DUBBLE OF /STUFF/
>20 PRINT "I AM ALL DONE"
>END
QUADRUBBLE DEFINED
←PRINT QUADRUBBLE OF "NO"
NONONONO
```

In the last example note that the command in line 20 was not executed. The OUTPUT in line 10 terminated the execution of QUADRUBBLE as it passed back "NONONONO" to PRINT. Thus we now have two means of terminating a procedure, either by the END command or the OUTPUT command.

2.7 Naming

A LOGO object can be given a name by use of the MAKE command. MAKE takes two inputs and makes the first the name of the second. Thus:

```
←MAKE "DIGIT" "TOE"
```

We say that "DIGIT" is the *name* of "TOE" and that "TOE" is the *thing* of "DIGIT". To retrieve the thing of a name, we use the one-input LOGO operation THING. Thus:

```
←PRINT THING OF "DIGIT"
TOE
```

So that a name refers to precisely one thing at any time, MAKE replaces the old thing of the name by the given new thing, the second input of MAKE. For example:

```
←MAKE (WORD OF "DIG" AND "IT") (SENTENCE OF "BIG" AND "TOE")
←PRINT THING OF "DIGIT"
BIG TOE
```

Note that the previous thing of "DIGIT", "TOE", has been replaced by "BIG TOE".

Initially, the empty thing, "", is taken as the thing of any LOGO name. Therefore:

```
←PRINT THING OF "ABRACADABRA"

←
```

(The thing "" has been printed.)

For introductory purposes, an extended form of MAKE is available. This is used by pressing the return key directly after typing in the word MAKE. LOGO will then type NAME: and wait for the user to type in the desired name and press the return key. LOGO then responds by typing THING:. After the user types the desired thing and presses the return key, the command is executed in exactly the same way as the standard form. Thus:

```
←MAKE
    NAME:    "1"
    THING:   "2"

←PRINT THING OF "1"
2
←
```

Because of the special importance of names, LOGO provides a shorthand notation for the operation THING. /IRVING/ means precisely the same as THING of "IRVING". Thus:

```
←PRINT SUM OF /1/ AND /1/
4
←
```

We have already encountered the use of slashes as delimiters for LOGO dummy inputs within procedure definitions. For example:

```
TO PYTHAGORAS /#/ /##/
10 PRINT SUM (PRODUCT /#/ /#/) (PRODUCT /##/ /##/)
END
```

When a procedure is executed, the occurrence of slashed objects in the title line results in implicit MAKE executions. Each is given as its thing the actual input in the corresponding position following the procedure name. For example, typing the command PYTHAGORAS 3 4 results in the name "#" being given the thing "3" and the name "##" being given the thing "4". /.../ means THING OF "..." whether the assignment was made by a procedure call or a MAKE command.

The names made by implicit MAKEs as a result of the execution of a title line have a special status. These names vanish from the list of names currently known to LOGO upon termination of the procedure. This is true even if the initial assignment of a thing to a dummy name is changed by a MAKE within the procedure.

```
TO SHOW-OFF /M/ /N/
10 PRINT SENTENCE OF "M IS" /M/
20 PRINT SENTENCE OF "N IS" /N/
END

←SHOW-OFF "TURTLE" "EGGS"
M IS TURTLE
N IS EGGS
←PRINT /M/

←PRINT /N/

←
```

If a dummy name is already known to LOGO, LOGO sets up a special version of the name expressly for use within the procedure. As before, each dummy name is given as its thing the corresponding input. Thus:

```
←MAKE "M" "IRVING"
←SHOW-OFF "EGGS" AND "TURTLE"
M IS EGGS
N IS TURTLE
←PRINT /M/
IRVING
←
```

Such special treatment is given to a dummy name independently in each procedure where it occurs. Thus:

```
TO SHO /A/
10 PRINT SENTENCE "/A/ IS" /A/
20 SHOO (BUTFIRST OF /A/)
30 PRINT [illegible]NTENCE "/A/ IS" /A/
END

TO SHOO /A/
10 PRINT SENTENCE "/A/ IS" /A/
20 SHOOO (BUTFIRST OF /A/)
30 PRINT SENTENCE "/A/ IS" /A/
END

TO SHOOO /A/
10 PRINT SENTENCE "/A/ IS" /A/
END

←SHO "START"
/A/ IS START                    (Printed by SHO.  At this point
                                 there is only one /A/)
/A/ IS TART                     (Printed by SHOO.  Now there are
                                 two versions of /A/)
/A/ IS ART                      (Printed by SHOOO.  Now there are
                                 three versions of /A/)
/A/ IS TART                     (Printed by SHOO.  Two versions of
                                /A/ remain since SHOOO has finished)
/A/ IS START                    (Printed by SHO.  Only one version
                                 of /A/ is left)
←                               (All versions of /A/ have vanished)
```

2.8 Conditional Operations

LOGO includes a number of operations called predicates, whose outputs are one of the words "TRUE" or "FALSE". NUMBERP is the LOGO number predicate.

```
←PRINT NUMBERP OF "777"
TRUE
←PRINT NUMBERP OF "SEVEN"
FALSE
←PRINT NUMBERP OF WORD OF 2 AND 2
TRUE
←
```

WORDP and SENTENCEP are the LOGO word predicate and LOGO sentence predicate:

```
←PRINT WORDP OF 711
TRUE
←PRINT WORDP OF "WORDS WORDS WORDS"
FALSE
←PRINT SENTENCEP OF "I C A R U S"
TRUE
←
```

Two useful predicates requiring two inputs are the identity predicate, IS, and the predicate GREATERP for comparing two numbers. IS outputs "TRUE" if its inputs are the same, and "FALSE" if they are different; GREATERP outputs "TRUE" if its first input is strictly greater than its second input, and "FALSE" otherwise.

```
←PRINT IS SUM OF 2 AND 2 "4"
TRUE
←PRINT IS SUM OF 2 AND 2 "04"
FALSE
←PRINT GREATERP 2 2
FALSE
←
```

Some predicates require one of the words "TRUE" or "FALSE" as inputs. These include the conjunctive predicate BOTH and the disjunctive predicate EITHER. BOTH outputs "TRUE" if both of its inputs are "TRUE"; EITHER outputs "TRUE" if either one or the other or both of its inputs are "TRUE".

```
←PRINT BOTH (IS WORD OF "2" AND "2" "22") AND (NUMBERP OF "SEVEN")
FALSE
←PRINT EITHER (GREATERP 2 3) AND (WORDP OF "FALSE")
TRUE
←
```

A predicate's output can be tested for "TRUE" or "FALSE" and subsequent instruction execution can be made to depend upon the result. LOGO provides a command, TEST, to facilitate such tests.

```
←TEST IS "1" "1"                (Clearly true)
←IF FALSE PRINT "1 IS NOT 1"    (So this line is not executed)
←IF TRUE PRINT "IT IS!"         (But, this one is)
IT IS!
←IF TRUE PRINT "YES YES YES"    (And this one also)
YES YES YES
←
```

TEST takes one input which must evaluate to "TRUE" or "FALSE". Its effect is to mark a "truth flag" correspondingly (i.e., to true or false). The associated command IF TRUE takes an instruction line as input and executes this line only if the truth flag is marked true. (Similarly with IF FALSE, when the truth flag is marked false.) Note that IF TRUE and IF FALSE take *command lines* as inputs -- in this respect they are different from other LOGO commands.

Using TEST and IF TRUE, new predicates can be defined by LOGO procedures. Thus:

```
←TO NOT /INPUT/
>10 TEST /INPUT/
>20 IF TRUE OUTPUT "FALSE"
>30 OUTPUT "TRUE"
>END
NOT DEFINED
←PRINT NOT IS "1" "2"
TRUE
←
```

2.9 Recursion

A procedure may use another procedure in its definition, as we saw in Section 2.6 where QUADRUBBLE was defined using DUBBLE. An even more powerful capability comes about by using a procedure in its own definition. An example of such a self-referential, or *recursive*, procedure is the following one, FIND, which outputs the /N/th element of /LIST/.

```
←TO FIND /LIST/ /N/
>10 TEST IS /N/ 1
>20 IF TRUE OUTPUT (FIRST OF /LIST/)
>30 OUTPUT FIND OF (BUTFIRST OF /LIST/)
     AND (DIFFERENCE OF /N/ AND 1)
>END

←PRINT FIND OF "Z Y X W" AND 1
Z
←PRINT FIND OF "Z Y X W" AND 3
X
←
```

FIND outputs immediately only in the case where /N/ is 1. It reduces all other cases to that one by creating a number of distinct copies of FIND. For the last example shown, three separate copies of FIND were used for execution of that instruction. The process was as follows:

The command line is:

```
     PRINT FIND "Z Y X W" 3
```

PRINT needs one input, so the execution goes to the next element on the line, which is FIND. Two literal inputs follow FIND, so FIND is executed line by line. Line 30 of FIND with the current inputs is

```
     30 OUTPUT FIND "Y X W" 2.
```

Thus, in order to finish execution of our original procedure FIND, we must perform FIND "Y X W" 2. A new copy of FIND, let us denote it (for ourselves) as FIND*, is used for this purpose and this copy is given inputs "Y X W" and 2. When line 30 of FIND* is reached, we have

```
     30 OUTPUT FIND "X W" 1.
```

Thus, still another copy of FIND is required, which we call FIND**, with inputs "X W" and 1. The execution of FIND** results in an output of "X" to FIND* (since the value of /N/ for this copy is 1) and FIND** "vanishes". FIND* now outputs "X" to FIND, which outputs "X" to PRINT.

We can display this sequence of successive procedure calls with the inputs and output associated with each of them in a compact way using the LOGO command TRACE. We indicate that we wish to TRACE the procedure FIND in its subsequent executions as follows.

```
←TRACE FIND
←                                (LOGO puts a trace on FIND and
                                  returns control to the user)
```

The effect of TRACE is illustrated next, using the example just discussed.

```
←PRINT FIND OF "Z Y X W" AND 3
FIND OF "Z Y X W" AND "3"
  FIND OF "Y X W" AND "2"        (This is our FIND*)
    FIND OF "X W" AND "1"        (Our FIND**, the third copy of FIND)
    FIND OUTPUTS "X"             (FIND** outputs to FIND*)
  FIND OUTPUTS "X"               (FIND* outputs to FIND)
FIND OUTPUTS "X"                 (FIND outputs to PRINT)
X                                (PRINT prints "X")
←
```

Note that TRACE prints the title line of each TRACEd procedure invoked, listing the inputs it is called with, and note that it prints a new line each time a procedure outputs or ends. The title line and output line of each procedure are indented the same number of spaces.

In the procedure FIND the execution of the command OUTPUT requires invoking and executing another copy of the procedure. In some recursive procedures the execution of other operations in the recursion line may be deferred as well. One such procedure is COLLAPSE, which takes a sentence and collapses it into a word.

```
←TO COLLAPSE /S/
>10 TEST IS /S/ /EMPTY/  (/EMPTY/ denotes the empty LOGO object, "")
>20 IF TRUE OUTPUT /EMPTY/
>30 OUTPUT WORD OF (FIRST OF /S/) AND
     (COLLAPSE OF BUTFIRST OF /S/)
>END
```

```
←PRINT COLLAPSE OF "CAN YOU READ ME"
CANYOUREADME
```

In the recursion line of COLLAPSE, line 30, the execution of the operation WORD, as well as the command OUTPUT, must be deferred. This is shown in the following trace.

```
←TRACE COLLAPSE
←PRINT COLLAPSE OF "MARES EAT OATS"
COLLAPSE OF "MARES EAT OATS"
  COLLAPSE OF "EAT OATS"
    COLLAPSE OF "OATS"
      COLLAPSE OF ""
      COLLAPSE OUTPUTS ""
    COLLAPSE OUTPUTS "OATS"
  COLLAPSE OUTPUTS "EATOATS"
COLLAPSE OUTPUTS "MARESEATOATS"
MARESEATOATS
←
```

The same form of recursion used with COLLAPSE was shown in Section 1 in the definitions of FACTORIAL and REVERSE. More complex and powerful forms of recursion can be created by the advanced user. These can even include recursions which are not reducible to iteration, such as the Ackerman function (the generalized exponential function used in recursive function theory).

3. Program Manipulation

Up to now we have studied only those parts of the LOGO language necessary for writing executable LOGO programs (the operations, commands, names, etc., and the rules governing their relations and usage). This chapter deals with those facilities of LOGO that aid a user in his programming work at the computer terminal. These include listing, editing, erasing, abbreviating, storing, and retrieving.

3.1 Editing

A. Editing a Line

There are a number of ways to modify the instruction line being typed in, at any time before the carriage return key is pressed. This "editing" capability works with both direct instruction lines and those which are part of a procedure definition.

If the backslash character "\" is pressed, this character is typed and its effect is to erase the character preceding it. The backslash can be typed more than once to effect multiple erasures. Thus:

```
←PINT \\\RINT 4\ 5
5
←PRIN T4\\\T 4
4
←
```

Pressing the CTRL key and W simultaneously results in a number of backslashes being typed, sufficient to completely erase the preceding LOGO word. Denoting this action by W^c, then:

```
←PRINT "THE QUIKW^C\\\\QUICK BROWN FOX"
THE QUICK BROWN FOX
←
```

The more drastic action of pressing the RUBOUT key erases the entire line. To show this has been done, the computer erases out the back arrow (←) preceding the "rubbed out" line, using a #. Denoting this editing action by (RUBOUT), then:

```
#PRINT "THE QUICK BROOW (RUBOUT)
←
```

B. Editing a Procedure

There are several editing commands which can be used only while a procedure is being defined. (The computer indicates that we are in the process of defining a procedure by typing a ">" rather than a "←" when it is ready to receive the next line.) The command EDIT, followed by a procedure name, is used to modify the definition of a previously defined procedure. LOGO responds to this command by typing a ">" to indicate that we are again in defining mode. After the desired changes have been made, the command END terminates the procedure definition just as before.

Inserting a Line

LOGO arranges the lines of a procedure in order of increasing line number. Thus, we can "insert" an instruction line between two already typed lines simply by giving it a number between the line numbers of the two given lines. It is good programming practice to number procedure lines by fives or by tens, to leave space for this possibility.

Assume we had previously defined REV:

```
TO REV /A/ /B/ /C/
10 PRINT /C/
20 PRINT /A/
END
```

and we wish to insert the instruction line PRINT /B/ between lines 10 and 20, as line 15, say. Then:

```
←EDIT REV
>15 PRINT /B/
>END
REV DEFINED
←
```

This effects the desired insertion.

Changing an Entire Line

We can change a previously entered instruction line by simply typing the desired instruction line, giving it the same number. The first version vanishes. We can retype the title line by typing TITLE, followed by the new title line.

Typing a line number and carriage return results in that line containing no instruction -- effectively erasing the line previously having that line number. A neater way to erase a line is by using the command ERASE LINE __, which completely expunges the line indicated.

The current version of any line can be shown by the LOGO command LIST LINE __, which prints out the indicated line. Similarly, LIST TITLE prints the title line.

Changing Part of a Line

Often it is easier to modify an existing line than to completely retype it. To do this, we type EDIT LINE ___, giving the appropriate line number. The computer will place the line number at the beginning of the next line. To have it type the next word of that line, we press the control key (CTRL) and N simultaneously. (This action is denoted by N^c.) To get the rest of the line, we type R^c (the control key and R). These two actions, together with "\" and W^c described above, can be used together, as in the following example:

```
>LIST LINE 30              (Note that we are already in defining mode,
30 PRONT SUM OF AND 2       the only context in which LIST LINE and
>EDIT LINE 30               EDIT LINE are meaningful)
30 N^CPRONTW^C\\\\\PRINT N^CSUM N^COF 3 R^C AND 2  (carriage return)
```

(Since N^c and R^c don't type out anything on the teletype, the above line looks readable.)

```
>LIST LINE 30
PRINT SUM OF 3 AND 2
```

Listing and Erasing the Entire Procedure

LIST (procedure name) results in the procedure being typed exactly as it stands at that moment. ERASE (procedure name) results in the procedure being expunged.

Nearly any LOGO instruction line including LIST (procedure name) and ERASE (procedure name) can be executed when in the procedure definition (>) mode. The only exceptions are those like TO and EDIT which involve the definition of yet another procedure while we are already defining one.

The following example shows a typical editing session involving the use of most of the features described in this section.

```
←LIST REVERSE
TO REVERSE /Y/                    (Should be /X/ in place of /Y/)
10 TEST IS /X/ /EMPTY/       (Missing is 15 IF TRUE OUTPUT /EMPTY/)
20 OUTPUT WORD (LAST /X/)
     (REVERS BUTLAST /X/)         (Incorrect spelling)
END
←EDIT REVERSE                (We could have used EDIT TITLE and then
>TITLE TO REVERSE /X/        NC twice to have LOGO type TITLE TO
>15 IF TRUE OUTPUT /EMPTY/   REVERSE)
>EDIT LINE 20
20 NCOUTPUT NCWORD NC(LAST NC/X/) NC
   (REVERS \E RCBUTLAST /X/)              (The \ deletes the space)
>LIST LINE 20
20 OUTPUT WORD (LAST /X/ (REVERSE
   BUTLAST /X/)
>LIST REVERSE
   (skips one line)
TO REVERSE /X/
10 TEST IS /X/ /EMPTY/
15 IF TRUE OUTPUT /EMPTY/
20 OUTPUT WORD (LAST /X/) (REVERSE
   BUTLAST /X/)
>END                              (Note there is no END command since
REVERSE DEFINED                   definition of reverse is not complete)
←
```

3.2 Abbreviating

To reduce the user's typing, the computer recognizes short forms for most commands. These are called abbreviations. For example:

```
←P S "CAT" "DOG"
CAT DOG
```

P is the abbreviation for PRINT and S for SENTENCE. The long forms are substituted internally for the abbreviations as soon as the abbreviations are typed in. Thus, if we type in a procedure definition making use of abbreviations and then list it, the computer types it back to us in expanded form. The set of built-in abbreviations is given as part of Section 5.

The user can make his own abbreviations with the command ABBREVIATE (two inputs). The first input can be any LOGO command, operation, or procedure, or combination of them. The second is the word which will become the abbreviation. The "noise word" AS may be inserted between the two inputs of ABBREVIATE.

```
←ABBREVIATE "PRINT SUM" AS "+"
←+ "3" "5"
8
←
```

An abbreviation can refer to just one operation. If we type:

```
←ABBREVIATE "RANDOM" "R"
←ABBREVIATE "REVERSE" "R"
```

the first meaning of "R" is lost.

Built-in abbreviations can also be changed.

`←ABBREVIATE "POWER" "P"` causes the abbreviation P for PRINT to vanish.

Listing and Erasing Abbreviations

LIST ALL ABBREVIATIONS results in the typing of all user-defined abbreviations, for example, we now have:

```
←LIST ALL ABBREVIATIONS

R:   REVERSE
P:   POWER
+:   PRINT SUM
←
```

ERASE ALL ABBREVIATIONS is used to erase all abbreviations, and ERASE ABBREVIATION (abbreviation) to erase the indicated abbreviation. For example,

```
←ERASE ABBREVIATION "+"
←+ 2 3
+ IS UNDEFINED
←
```

3.3 The User Workspace

Upon logging in on a computer and requesting LOGO, the user has at his disposal all built-in features of the LOGO language. These include the LOGO operations and commands, reserved names, such as /EMPTY/, and standard abbreviations. The user is also assigned a *workspace* within the computer memory. The additions he makes to the LOGO built-ins are kept in this workspace. These possible additions include user-defined procedures, user-defined abbreviations, and those user-defined names which were not created by execution of procedure title lines. Each class of objects in the user workspace can be listed or erased separately:

```
LIST ALL PROCEDURES
ERASE ALL PROCEDURES

LIST ALL NAMES
ERASE ALL NAMES

LIST ALL ABBREVIATIONS
ERASE ALL ABBREVIATIONS
```

Abbreviations are the only built-ins which can be changed by the user. ERASE ALL ABBREVIATIONS not only erases all user abbreviations, but restores the built-in abbreviations to their original state.

The commands LIST ALL and ERASE ALL combine the listing and erasing commands for all these three types of objects in user workspace. Thus, LIST ALL provides an exact accounting of everything in the user workspace and ERASE ALL completely empties the workspace. Any procedure, name, or abbreviation can be listed or erased individually, as described in preceding sections.

The command LIST CONTENTS lists just the title line of every procedure in workspace. For example:

```
←LIST CONTENTS

TO REVERSE /X/
TO FACTORIAL /N/
←
```

GOODBYE, as well as exiting from LOGO, results in the complete loss of the user workspace. LOGO provides commands to save the contents of the workspace for subsequent use. If such retention is desired, it must be effected before GOODBYE is typed. The saved material can then be retrieved at any later time. This process of "SAVEing" and "GETting" is described next.

3.4 Filing

LOGO provides a facility for users to file away their work. The basic unit of a LOGO file is an *entry*. Each entry has a two-word *entry name*. The first word of the entry name is the *file name* and is common to all the entries in a file (it is commonly the name of the user who owns the file). The second word of the entry name distinguishes the entry from other entries in the same file. Examples of entry names are JIM EQUATIONS, NANCY RANDOMSENT, JIM NIM, NANCY EQUATIONS.

An entry is created by the command SAVE. The entry thus created contains everything in the user workspace -- that is, everything that would be listed by LIST ALL. The user workspace is left unchanged by the SAVE command. For example,

```
←SAVE GRANT ARITH
←
```

In this example, the entry GRANT ARITH is created. If GRANT ARITH already exists, the old entry is replaced by the new one. Although the user workspace may subsequently change, the contents of the entry GRANT ARITH will remain as they were at the time they were saved. When the user gives the command GOODBYE, the workspace is destroyed but all entries are retained.

In a well-organized file, each entry contains a related group of procedures, names, and abbreviations (for example, those that are used for playing NIM, or those used in solving linear equations). By first erasing irrelevant parts, the user can save any desired subset of his workspace.

To retrieve an entry from a file, the command GET is used.

```
←GET GRANT ARITH
←
```

The contents are copied into, and become a part of, the student's current workspace -- the entry itself is unchanged. The additions to the workspace provided by a GET are inserted in the same way as if a user had typed them in. Thus, abbreviations and names supersede existing ones and procedures in the entry having the same names as those in the workspace are not entered.

There are three different types of objects possible in the user workspace, and hence in any entry. These parts of an entry can be listed by the commands

```
    LIST PROCEDURES _____ _____         (where the dashes
    LIST NAMES _____ _____              indicate the two-word
    LIST ABBREVIATIONS _____ _____      entry name)
```

LIST ENTRY ____ ____ lists everything in the entry -- all three parts. LIST CONTENTS ____ ____ gives the title line of each procedure contained in the entry indicated.

LISTing any part of an entry does not result in its being copied into the active workspace. Only GET will do this.

The command LIST ALL FILES causes LOGO to type all existing file names. The command LIST FILE ____ causes LOGO to type the entries in a given file. To remove an entry from a file, the command ERASE ENTRY is used:

```
←ERASE ENTRY GRANT ARITH
←GET GRANT ARITH
THERE IS NO ENTRY GRANT ARITH
←
```

When all entries in a file are erased, the file itself is automatically eliminated.

LOGO also provides two *operations* for working with LOGO files.

SIZE ____ ____ outputs a number proportional to the amount of space the entry indicated occupies in memory.

ENTRIES ____ ____ outputs a sentence of the second words of all entry names contained in the file indicated.

3.5 Debugging

The LOGO system has built-in aids to help users find the "bugs" in their programs. A bug has one of two effects. It may cause the computer to try to execute an illegal instruction or it may direct the execution of instructions that are legal but which produce a wrong answer or no answer at all, e.g., it may put the computer in a loop that never ends.

In the first case, the computer immediately stops executing instructions and types out a diagnostic message describing the error and telling where it occurred. (Some typical diagnostic messages are listed at the end of this section.) An example of this is:

```
←TO GREET /X/
>10 PRINT SENTENCE OF "HELLO," AND /X/
>20 PRONT "HOW ARE YOU?"
>30 PRINT "SEE YOU LATER"
>END
GREET DEFINED
```

```
←GREET "JOHN"
HELLO, JOHN

PRONT NEEDS A MEANING.
I WAS AT LINE 20 IN GREET
```

There was a bug. The diagnostic message designates the type of error and where the error was found. EDIT can be used to make the necessary changes.

```
←EDIT GREET
>20 PRINT "HOW ARE YOU?"
>END
GREET DEFINED

←GREET "JOHN"
HELLO, JOHN
HOW ARE YOU?
SEE YOU LATER
←
```

When the procedure GREET was being defined, the computer didn't object when line 20 was typed in, even though it did not know the meaning of PRONT. The reason is that a procedure PRONT might have been written later, after GREET was defined but before it was executed.

In this example, the computer's diagnostic message pointed to the source of the error and thus was directly helpful. Often, however, we get situations where the illegal instruction isn't the direct cause of the error. For example, in the course of running a procedure the computer may say

```
DIFFERENCE OF "AB" AND "1"?  INPUTS MUST BE NUMBERS.
I WAS AT LINE 30 OF SAM.

←EDIT SAM
>LIST LINE 30
30 OUTPUT SUM OF /X/ AND PRODUCT OF /X/ AND DIFFERENCE OF
  /Y/ AND "1"
```

Assuming the arithmetic expression given is the one intended, the error is not contained in line 30. Somewhere earlier in the execution, /Y/ was made "AB" instead of a number. This type of error then is of the second type mentioned above. The computer gets past the faulty instruction and the defective result shows up as a bug later when the computer is performing another instruction, perhaps in a different procedure. In this situation the diagnostic is less helpful and the error is more difficult to track down.

The TRACE command, described in the section on recursion, is often useful in finding errors, especially those resulting in a faulty recursion. TRACE, followed by a procedure name, results in a special "flag" being placed with the procedure. Then, whenever the procedure is called in an execution, its title line is typed with the current values of its dummy variables. When the procedure outputs, or is otherwise completed, an appropriate typeout is made and the output, if any, is shown. If a procedure is in TRACE mode, this is indicated whenever the procedure is LISTed. To get out of TRACE mode, the command ERASE TRACE, followed by the procedure name, is used. ERASE ALL TRACES is a more drastic LOGO command.

Another useful approach, when the difficulty lies within a known procedure rather than "between" procedures, is to insert extra lines in the defective procedure to type intermediate results. These help to pinpoint the error and can be removed after a correction has been made.

Diagnostic Messages

There are about 100 diagnostic messages. The following are some typical ones.

```
YOU NEED / MARKS AROUND EACH INPUT.
TITLE MUST BE FOLLOWED BY "TO".
END WHAT?  YOU'RE NOT DEFINING ANYTHING.
GO WHERE?
LIST ALL WHAT?
DON'T TRY TO DEFINE ANOTHER PROCEDURE INSIDE THIS ONE.
DIVISION BY ZERO.
DON'T USE THE EMPTY WORD FOR A NAME.
ERASE WHAT?
THE INPUT TO TEST MUST BE A PREDICATE.
```

The following comments mean that the number of inputs found on the line was not correct. The exact form of comment depends on the particular parsing error.

```
____ IS EXTRA
THERE ARE _____ INPUTS MISSING FOR ____
```

In the following diagnostics, the underscored words are filled in appropriately by LOGO when the error occurs. The words given here are typical examples.

```
MATCHING"?  (or / or ( or ))
PRONT NEEDS A MEANING.
TRUMP ISN'T COMPLETELY DEFINED.    (END command not yet given.)
```

```
THERE IS NO LINE 30.
SUM OF "A" AND "5"?  INPUTS MUST BE NUMBERS.
TEST IS USED BY LOGO.           (The user cannot define a procedure
                                 called TEST)
OF ISN'T A PROCEDURE.
THERE ISN'T ANY FILE GRANT
REVERSE IS ALREADY DEFINED.
YOU'RE ALREADY DEFINING REVERSE.
YOU'RE ALREADY EDITING REVERSE.
REVERSE CAN'T BE USED AS AN INPUT.  IT DOESN'T OUTPUT.
```

The comment I AM IN TROUBLE. TELL YOUR TEACHER indicates a computer failure.

3.6 Interrupting Execution

The execution of a direct line or procedure is interrupted by the momentary depression of the key labeled BREAK. The pressing of the BREAK key is effective, whether the computer is performing internal operations or printing on the teletypewriter. When this occurs, LOGO types "BREAK" as well as the procedure name and line it was then executing, and then returns control to the user. (This "positional" information is omitted if a direct line was interrupted.) The state of the execution is preserved -- all intermediate results are kept. These include all the local names set up by the use of dummy variables in procedures which had not yet terminated. LIST ALL NAMES gives all these "local" names in order opposite to the order of their creation. Control has returned to the user exactly as though these intermediate results did not exist. They do not get saved by a SAVE command, nor do they interfere with any procedure definition or execution. They can, however, slow down execution somewhat because they take up room in the user's workspace. The only real effect these intermediate results have is initiated by the no-input command GO. This results in the interrupted calculation being resumed exactly from the point left off. The only loss that can occur is that of some printing that was in process when BREAK was pressed. Any changes made in the interim will, of course, result in a continuation different from that produced if no interruption had taken place. Thus, the BREAK key can be a useful debugging tool.

The no-input command CANCEL erases the intermediate results produced by the calculation interrupted by the latest BREAK. Thus, if an execution has been interrupted, another initiated without the use of GO and the BREAK key again pressed, two uses of CANCEL are needed to erase all the intermediate results existing in the workspace. It is good practice to use CANCEL after interrupting any procedure which is not to be resumed using GO.

4. Additional Commands and Operations

4.1 Formatting

There is one command and several reserved LOGO names for positioning characters to be typed. They are particularly useful for writing procedures to "draw" figures on the teletype, but have other uses as well.

TYPE is a command which causes its one input to be typed. It differs from PRINT only in that the typehead remains positioned just after the character last typed -- there is no return to the beginning of the next line.

Some special names are used for formatting with a PRINT or TYPE command. They are:

/BLANK/	a blank space. (the space bar cannot be used for this because LOGO eliminates superfluous spaces.)
/LINE FEED/	teletype goes to next line without moving horizontally.
/CARRIAGE RETURN/	return to beginning of current line.
/SKIP/	a new line. (has the effect of both a carriage return and a line feed.)
/FORM FEED/	move paper to new page on teletypes which have a form feed feature.

Then, to have the computer draw a diagonal line:

```
←TO DIAGONAL /N/
>10 TEST IS /N/ 0                 (Are we finished?)
>20 IF TRUE STOP
>30 TYPE ":"
>40 TYPE /BLANK/                  (Move across one)
>50 TYPE /LINE FEED/              (Move down one)
>60 DIAGONAL (DIFF /N/ 1)         (Repeat for /N/ -1)
>END
DIAGONAL DEFINED
←DIAGONAL 8
:
 :
  :
   :
    :
     :
      :
       :
←
```

4.2 Interactive Programs

All commands and operations used thus far must have their inputs specified before they are executed. Each such input is a literal, the thing of a name, or the output of some operation or procedure. The LOGO operation REQUEST, however, causes execution to pause until the user has typed a string of characters and a carriage return. REQUEST then outputs this string. For example:

```
←PRINT REQUEST
*HUMBUG                 (REQUEST prints an asterisk "*" to show
HUMBUG                  that user type-in is required*)
←PRINT SUM OF REQUEST AND REQUEST
*3                      (The leftmost REQUEST)
*2
5
←
```

REQUEST makes possible the writing of programs which "interact" with the user.

```
←TO COPYCAT
>10 PRINT "TELL ME SOMETHING."
>20 PRINT REQUEST
>30 COPYCAT
>END
COPYCAT DEFINED
←COPYCAT
TELL ME SOMETHING.
*WHO ARE YOU?
WHO ARE YOU?
TELL ME SOMETHING.
*WHY SHOULD I?
WHY SHOULD I?
TELL ME SOMETHING.
*ARE YOU SOME KIND OF NUT
ARE YOU SOME KIND OF NUT
TELL ME SOMETHING.
*

  .  .  .
  .  .  .
  .  .  .
```

---

* This asterisk is omitted when LOGO is not at the left-hand edge of the paper. This is often the case when the last typing resulted from a TYPE command.

The existence of an interactive capability makes the element of time particularly interesting. There are several ways in which LOGO makes provision for timing.

The operation ASK requires one input which must be a number. ASK is the same as REQUEST unless the user has not completed his typing when a number of seconds equal to the input has elapsed. If this happens, ASK outputs the empty word and returns the carriage to a new line.

```
←TO QUICKQUERY /QUESTION/
>10 PRINT /QUESTION/
>20 MAKE "ANSWER" ASK 5
>30 TEST IS /ANSWER/ /EMPTY/
>40 IF TRUE PRINT "YOU WEREN'T FAST ENOUGH"
>50 OUTPUT /ANSWER/
>END
QUICKQUERY DEFINED
←PRINT QUICKQUERY "WHO DISCOVERED FERMAT'S LAST THEOREM?
  YOU HAVE 5 SECONDS TO ANSWER."
::FERM         (If as here, 5 seconds have elapsed before the user
               presses CARRIAGE RETURN, [illegible]O resumes control)
YOU WEREN'T FAST ENOUGH
               (The empty line is printed here by QUICKQUERY)
←
```

The current date and time are made available to LOGO by the no-input operations DATE and TIME:

```
←PRINT DATE
6/13/71
←PRINT TIME
11:05 PM
```

There is also an internal "clock" which is started when the user enters LOGO. This clock keeps time in seconds:

```
←PRINT CLOCK
1806
←PRINT CLOCK
1811
```

RESET CLOCK sets the clock back to 0.

```
←PRINT CLOCK
1825
←RESET CLOCK
←PRINT CLOCK
5
←
```

The WAIT command makes LOGO pause a number of seconds equal to its one input. It has no other effect. For auditory interaction, there is the reserved name "BELL". PRINT /BELL/ rings the teletype bell.

4.3 More Arithmetic

There are several numerical operations besides the basic four operations and GREATERP and RANDOM, all discussed earlier. All built-in numerical operations require integer inputs.

The operation QUOTIENT simply outputs the integer part of the quotient of its two inputs. REMAINDER outputs the remainder of the division yielding the quotient. DIVISION outputs a sentence of two numbers -- the quotient of its two inputs and the remainder.

```
←PRINT QUOTIENT OF 34 AND -6
-5
←PRINT REMAINDER OF 34 AND -6
4
←PRINT DIVISION OF 34 AND -6
-5 4
←
```

MAXIMUM outputs the greater of its two inputs. MINIMUM outputs the lesser of its two inputs.

```
←TO ORDER2 /A/ /B/
>10 OUTPUT SENTENCE
     MINIMUM /A/ /B/
     MAXIMUM /A/ /B/
>END
ORDER2 DEFINED
←PRINT ORDER2 3 -1
-1 3
←
```

ZEROP is a one-input operation which outputs "TRUE" or "FALSE" as the input is, or is not, numerically equal to zero. Thus, ZEROP is not the same as IS 0.

```
←PRINT IS 0 00
FALSE
←PRINT ZEROP 00
TRUE
←
```

COUNT is not itself a numerical operation, but it has an integer output, so it usually appears in conjunction with numerical operations. COUNT has one input. Its output is the number of letters in the input, if it is a LOGO word -- or the number of words, if it is a LOGO sentence. Thus:

```
←PRINT COUNT OF "ABC"
3
←PRINT COUNT OF "THE CAT IN THE HAT"
5
←PRINT COUNT ""
0
←
```

4.4 Local and Global Names

The Command LOCAL

As we saw earlier, including a name on the title line of a procedure meant that a special copy of the name would be created each time the procedure was invoked. Each copy disappears when the procedure which created it terminates. This feature is especially useful when a procedure makes copies of itself recursively, like the procedure REVERSE in the section dealing with recursion. When a name does not appear in the title line of the procedure in which it is used, no special copy of the name is made. This is often a useful feature when we use such a *global name* to transfer information from one procedure to another. Sometimes, as in the example following, a "slight" variation of REVERSE, it is a handicap.

```
←TO REVERSE /INPUT/
>10 TEST IS /INPUT/ /EMPTY/
>20 IF TRUE OUTPUT /EMPTY/
>30 MAKE "Y" FIRST OF /INPUT/
>40 OUTPUT WORD OF
       REVERSE (BUTFIRST OF /INPUT/) AND /Y/
>END
REVERSE DEFINED
←PRINT REVERSE "HELLO"
OOOOO
```

```
←TRACE REVERSE
←PRINT REVERSE OF "HELLO"
REVERSE OF "HELLO"                (/Y/ is now "H")
  REVERSE OF "ELLO"               (/Y/ is now "E")
    REVERSE OF "LLO"              (/Y/ is now "L")
      REVERSE OF "LO"             (/Y/ is now "L")
        REVERSE OF "O"            (/Y/ is now "O")
          REVERSE OF ""           (Since input is /EMPTY/, /Y/ is
          REVERSE OUTPUTS ""       not changed)
        REVERSE OUTPUTS "O"
      REVERSE OUTPUTS "OO"
    REVERSE OUTPUTS "OOO"
  REVERSE OUTPUTS "OOOO"
REVERSE OUTPUTS "OOOOO"
OOOOO                             (The result is PRINTed)
←
```

"Y" is made "O" in the fifth copy of REVERSE and never changes thereafter. In the succeeding outputs, this /Y/ is what is actually used. We really intended that a new copy of "Y" exist for each calling of REVERSE. We can easily accomplish this by the insertion of the instruction:

```
5 LOCAL "Y"
```

This results in "Y" being handled just as if it were on the title line. Now REVERSE works.

```
←PRINT REVERSE "HELLO"
OLLEH
←
```

LOCAL is an unusual command in that it allows any number of inputs to follow it. Each is taken as a name to be made "local" to the procedure in which LOCAL appears.

4.5 Automatic Program Generation - An Advanced Feature

DO is a LOGO command which results in the execution of its one input. Thus, for example,

```
←DO "EDIT FOO"
>                   (and we are editing FOO)
```

In the case above we could just as well have typed in EDIT FOO directly, omitting the quotes and the DO. But, we could also have written, in still larger form:

```
←MAKE "PNAME" "FOO"
←DO SENTENCE "EDIT" AND /PNAME/
```

Here we see the utility of DO. It enables exact specification of parts of statements to be deferred, which otherwise would have had to be inserted in literal form. Thus the command DO forms the basis for general procedures which create or modify other procedures.

Two operations are provided by LOGO to enable a procedure to find the current contents of a procedure to be modified.

LINES is a one-input procedure. It outputs the sentence of the line numbers of the procedure given as input.

TEXT is a two-input procedure -- it requires a procedure name and a line number. It outputs the entire line, as a sentence. If the line number 0 is given, TEXT outputs the title line of the procedure indicated.

Thus, LINES can be used to find what lines exist and TEXT to go through them one-by-one.

To illustrate these commands, consider the following procedure for replacing /WORD/ by /SUBST/ in /SENTENCE/.

```
←TO REPLACE /WORD/ /SUBST/ /SENTENCE/
>10 TEST IS /SENTENCE/ /EMPTY/
>20 IF TRUE OUTPUT /EMPTY/
>30 TEST IS (FIRST /SENTENCE/) /WORD/
>40 IF TRUE OUTPUT SENTENCE
      /SUBST/
      REPLACE /WORD/ /SUBST/ (BUTFIRST /SENTENCE/)
>50 OUTPUT SENTENCE
      FIRST /SENTENCE/
      REPLACE /WORD/ /SUBST/ (BUTFIRST /SENTENCE/)
>END
REPLACE DEFINED
←PRINT REPLACE OF "CATS" "DOGS" "IT'S RAINING CATS AND DOGS"
IT'S RAINING DOGS AND DOGS
←PRINT LINES OF "REPLACE"
10 20 30 40 50
←PRINT TEXT OF "REPLACE" AND 30
30 TEST IS (FIRST /SENTENCE/) /WORD/
←
```

The following set of procedures, using the procedure REPLACE, inserts /W2/ in place of /W1/ everywhere the latter appears in a procedure /PNAME/. It exemplifies the use of DO, LINES, and TEXT.

```
TO MODIFY /PNAME/ /W1/ /W2/
10 DO SENTENCE OF "EDIT" AND /PNAME/
20 CHANGE (LINES OF /PNAME/) /W1/ /W2/
30 DO "END"
END
```

```
TO CHANGE /LINES/ /W1/ /W2/
10 TEST IS /LINES/ /EMPTY/
20 IF TRUE STOP
30 DO SENTENCE SENTENCE SENTENCE
     "REPLACE"
     /W1/
     /W2/
     TEXT OF (FIRST /LINES/)
40 CHANGE (BUTFIRST OF /LINES/) /W1/ /W2/
END
```

The reserved name "CONTENTS" has as its thing, the sentence of all procedure names (not title lines) in workspace. This makes possible writing of procedures even more general than the above. For further generality, /FILES/ is a sentence consisting of all file names.

4.6 Other Ways to Terminate a Procedure

To stop execution of a procedure before the END command, the 0 input command STOP may be used. Its effect is exactly that of END -- the procedure simply stops and control returns to whatever called the procedure. STOP is often used to terminate one branch resulting from a TEST. For example,

```
TO FACTOR /A/
10 TEST ZEROP /A/
20 IF TRUE PRINT "I CANNOT FACTOR ZERO"
30 IF TRUE STOP
 :  :  :
```

The one-input command EXIT also terminates a procedure. Its input is typed, then LOGO acts exactly as though an error has been found, typing the location of the EXIT command and returning control to the user. Thus, the example of a partial procedure preceding could have been written

```
TO FACTOR /A/
10 TEST ZEROP /A/
20 IF TRUE EXIT "I CANNOT FACTOR ZERO"
 :  :  :
```

and, after the procedure has been completed,

```
←FACTOR 0
I CANNOT FACTOR ZERO
I WAS AT LINE 20 OF FACTOR
```

4.7 Miscellany

The Turtle

There is a set of operations and commands reserved for the "turtle", a LOGO-controlled robot.

The 0-input commands FRONT and BACK move the turtle one unit in the directions they name. RIGHT and LEFT rotate the turtle clockwise and counterclockwise. HORN rings the turtle's bell.

The 0-input operations TOUCH LEFT and TOUCH RIGHT refer to the turtle's touch sensors. TOUCH LEFT outputs TRUE if the left sensor is against an obstacle, otherwise, it outputs FALSE. TOUCH RIGHT queries the right touch sensor in the same way.

Other Commands and Operations

EMPTYP is a one-input operation which outputs TRUE or FALSE as its input is or is not the empty word. EMPTYP has exactly the same effect as IS /EMPTY/.

IGNORE is a one-input command which has no effect. It is used in the rare situations where an output which has no further use is generated. For example, the lines

```
50 PRINT "PRESS CARRIAGE RETURN TO CONTINUE"
60 IGNORE REQUEST
```

result in typing of the message given by line 50, followed by typing of "*" and a pause in execution until the CARRIAGE RETURN key is pressed.

GO TO LINE is a one-input command which is valid only in a procedure. It causes execution to pass to the line whose number is given as its input.

Comments

A user may place remarks which he does not wish to be executed anywhere within a procedure definition. The user indicates that a string is not to be executed by placing semicolons around it.* The only restriction on inserting comments in this way is that they may not be placed within the quotes demarcating a literal or within the pair of slashes delimiting a LOGO name. Remarks correctly indicated have no effect on the execution of the procedure they lie within. They only appear when the procedure is listed. For example,

```
TO DIAGONAL /N/; DRAWS A DIAGONAL LINE*
10 TEST IS /N/ 0; END TEST
20 ; IF /N/ IS 0 WE ARE DONE; IF TRUE STOP
30 TYPE "*"
40 TYPE /BLANK/; MOVE ACROSS ONE SPACE
50 TYPE /LINE FEED/; NEXT LINE
60 ; REPEAT FOR /N/ -1; DIAGONAL (DIFF /N/ 1)
END
```

Comments may be placed after the entry name in a SAVE command, again preceded by a semicolon. The comment is typed whenever the entry name appears in a listing:

```
←SAVE GRANT ARITH; A GENERAL ARITHMETIC PACKAGE
```

/QUOTE/

If the user tries to print "DOG", complete with quotes, he gets an error.

```
←PRINT ""DOG""
                   (prints empty word)
DOG"" IS EXTRA
```

The special LOGO thing /QUOTE/ is used in such a situation to indicate a quote mark which is not intended as the delimiter of a literal. Thus:

```
←PRINT SENTENCE SENTENCE /QUOTE/ "DOG" /QUOTE/
"DOG"
```

---

*A comment at the end of a line need not be terminated with a semicolon.

5. Glossary and Index

	Abbr.	Description	Page
ABBREVIATE	ABT	(2-input command) sets up second input as the abbreviation of the first input	23
AND		("noise" word) used for clarity; valid only between inputs of a procedure	6
AS		("noise" word) valid only in abbreviating	23
ASK		(1-input operation) outputs literal type-in from teletype if completed in input number of seconds, else the empty word	33
BACK		(0-input command) "turtle" effector, moves turtle back 1 space	39
BOTH	B	(2-input operation) each input must be TRUE or FALSE; outputs TRUE if both are TRUE, otherwise FALSE	16
BUTFIRST	BF	(1-input operation) outputs all but first character of an input word, or all but first word of input sentence	7
BUTLAST	BL	(1-input operation) outputs all but last character of input word, or all but last word of input sentence	7
CANCEL		(0-input command) eliminates one level of break	30
CLOCK		(0-input operation) outputs time given by internal one-second clock	33
COUNT	C	(1-input operation) outputs number of characters of an input word, or number of words of an input sentence	35
DATE		(0-input operation) outputs current date	33
DIFFERENCE	DIFF	(2-input operation) difference of first and second input, which must be integers	8
DIVISION	DIV	(2-input operation) inputs must be integers. Output is sentence of integer quotient and remainder	34
DO		(1-input command) executes its input as a LOGO instruction line	36
EDIT		(command followed by procedure name) puts LOGO into define mode	21,23,28
EDIT LINE	EDL	(1-input command) used to edit line indicated, valid only in define mode	22,23

	Abbr.	Description	Page
EDIT TITLE	EDT	(0-input command) used to edit title, valid only in define mode	23
EITHER	EI	(2-input operation) each input must be TRUE or FALSE. Outputs TRUE if either input is TRUE, otherwise FALSE	16
EMPTYP	EP	(1-input operation) outputs TRUE if input is the empty word, otherwise FALSE	39
END		(0-input command) terminates a procedure definition	5
ENTRIES		(1-input operation) outputs sentence of second words of entry names in file given as input	27
ERASE	ER	(command, followed by procedure name) erases procedure from workspace	22
ERASE ABBREVIATION		(1-input command) erases abbreviation given as input	24
ERASE ALL		(0-input command) completely erases workspace, restores built-in abbreviations	25
ERASE ALL ABBREVIATIONS		(0-input command) erases all user-defined abbreviations in workspace	24,25
ERASE ALL NAMES		(0-input command) erases all user-defined names in workspace	25
ERASE ALL PROCEDURES		(0-input command) erases all procedures from workspace	25
ERASE ALL TRACES		(0-input command) removes trace flag from all traced procedures in workspace	29
ERASE ENTRY	EE	(command, followed by entry name) completely erases indicated entry	27
ERASE LINE	ERL	(1-input command) only valid in define mode. Erases line whose number is given as input	21
ERASE TRACE		(command, followed by procedure name) removes trace flag from indicated procedure	29
EXIT		(1-input command) types its input, the line number and procedure name in which it appears and terminates execution	38
FIRST	F	(1-input operation) outputs the first character of an input word, or the first word of an input sentence	7

	Abbr.	Description	Page
FRONT		(0-input command) "turtle" effector, moves turtle forward 1 space	39
GET		(command, followed by entry name) enters indicated entry into workspace	26
GO		(0-input command) continues execution from a BREAK key interrupt	30
GO TO LINE	GTL	(1-input command) only valid within a procedure definition. Transfers execution to line whose number is given as input	40
GOODBYE	GB	(0-input command) terminates LOGO session	3,25
GREATERP	GP	(2-input operation) inputs must be integers. Outputs TRUE if first input is strictly greater than second, else FALSE	16
HORN		(0-input command) "turtle" effector, rings turtle's bell	39
IF FALSE	IFF	(command followed by instruction) executes instruction if truth flag is FALSE, otherwise has no effect	16
IF TRUE	IFT	(command followed by instruction) executes instruction if truth flag is TRUE, otherwise has no effect	16
IGNORE		(1-input command) has no effect	39
IS		(2-input operation) outputs TRUE if first input is identical to second, otherwise FALSE	16
LAST	L	(1-input operation) outputs last character of an input word, or last word of input sentence	7
LEFT		(0-input command) "turtle" effector, rotates turtle counterclockwise	39
LINES		(1-input operation) output is sentence of all instruction line numbers of procedure whose name is given as input	37
LIST		(command followed by procedure name) types definition of indicated procedure	22
LIST ABBREVIATIONS		(command followed by entry name) types all abbreviations in entry indicated	26
LIST ALL		(0-input command) types entire user workspace	25

	Abbr.	Description	Page
LIST ALL ABBREVIATIONS		(0-input command) types all abbreviations in user workspace	24
LIST ALL FILES		(0-input command) types all file names	27
LIST ALL NAMES		(0-input command) types all names in user workspace	25,30
LIST ALL PROCEDURES		(0-input command) types definitions of all procedures in user workspace	25
LIST CONTENTS	LC	(0-input command) types title lines of all procedures in user workspace	25
LIST CONTENTS	LC	(command followed by entry name) types title lines of all procedures in entry indicated	27
LIST ENTRY	LE	(command followed by entry name) types entire entry indicated	27
LIST FILE		(command followed by file name) types list of entries in file indicated	27
LIST LINE	LL	(1-input command) valid only in define mode. Types line whose number is given as input	22,23
LIST NAMES		(command followed by entry name) types all names in entry indicated	26
LIST PROCEDURES		(command followed by entry name) types all procedure definitions in entry indicated	26
LIST TITLE		(0-input command) valid only in define mode. Types title line of procedure being defined	22
LOCAL		(command with any number of inputs) only valid in a procedure. Makes all names given as inputs local to the procedure containing this command	35
MAKE		(2-input command) makes the first input the name of the second input	12
MAXIMUM	MAX	(2-input operation) inputs must be integers. Outputs the greater of the inputs	34
MINIMUM	MIN	(2-input operation) inputs must be integers. Outputs the lesser of the inputs	34
NUMBERP	NP	(1-input operation) outputs TRUE if input is an integer, otherwise FALSE	15

	Abbr.	Description	Page
OF		("noise" word) used for clarity; only valid following procedure name	6
OUTPUT	OP	(1-input command) valid only in procedure. Passes its input to procedure (or operation or command) which called current procedure	11
PRINT	P	(1-input command) types its input on teletype and returns carriage to beginning of next line	3
PRODUCT	PROD	(2-input operation) inputs must be integers. Outputs their product	6,8
QUOTIENT	QUO	(2-input operation) inputs must be integers. Outputs their integer quotient	8,34
RANDOM		(0-input operation) outputs a random digit	8
REMAINDER	REM	(2-input operation) inputs must be integers. Outputs remainder of their integer division	34
REQUEST	RQ	(0-input operation) outputs literal type-in from teletype	32
RESET CLOCK		(0-input command) resets internal clock to zero	33
RIGHT		(0-input command) "turtle" effector, rotates turtle clockwise	39
SAVE		(command followed by entry name) saves user workspace as entry with name indicated	26
SENTENCE	S	(2-input operation) outputs sentence of its inputs	7
SENTENCEP	SP	(1-input operation) outputs TRUE if input is a sentence, otherwise FALSE	15
SIZE		(1-input operation) outputs "size" of entry given as input	27
STOP		(0-input command) valid only in procedure. Terminates execution of its procedure	38
SUM		(2-input operation) inputs must be integers. Outputs their sum	8
TEST	T	(1-input command) input must be TRUE or FALSE. Sets truth flag to its input	16

	Abbr.	Description	Page
TEXT		(2-input operation) outputs text in procedure given by first input, with line number given by second input	37
THING		(1-input operation) outputs thing named by the input	12
TIME		(0-input operation) outputs current time	33
TITLE		(command followed by title line) valid only in define mode. Changes title of procedure being defined to that following	21
TO		(command followed by title line) enters define mode of procedure name following	4
TOUCH LEFT		(0-input operation) "turtle" feedback, outputs TRUE if left sensor has touched obstacle, resets touch flag, otherwise outputs FALSE	39
TOUCH RIGHT		(0-input operation) "turtle" feedback, outputs TRUE if right sensor has touched obstacle, resets touch flag, otherwise outputs FALSE	39
TRACE		(command followed by procedure name) sets trace flag for procedure indicated	18,29,36
TYPE		(1-input command) types its input	31
WAIT		(1-input command) causes execution to pause a number of seconds equal to its input	34
WORD	W	(2-input operation) inputs must be words. Outputs the word formed by concatenating them	7
WORDP	WP	(1-input operation) outputs TRUE if input is a word, otherwise FALSE	15
ZEROP	ZP	(1-input operation) outputs TRUE if input is equal to zero, otherwise FALSE	34

There are also several abbreviations for parts of commands. They are

ABB: ABBREVIATION
ABBS: ABBREVIATIONS
ER: ERASE
PRS: PROCEDURES

Reserved Names

	Description	Page
/EMPTY/	the empty thing	19
/CONTENTS/	a sentence of user-defined procedure names	38
/LINE FEED/	a line feed without carriage return when typed	31
/CARRIAGE RETURN/	a carriage return without line feed when typed	31
/FILES/	a sentence of file names	38
/FORM FEED/	moves paper to a new page when typed on teletypes having form feed feature	31
/BLANK/	a blank space when typed	31
/BELL/	rings a bell when typed	34
/QUOTE/	a quote mark	40
/SKIP/	a new line (carriage return and line feed) when typed	31

Special Keys

RETURN	gives line just typed by user to LOGO	3
RUB OUT	erases line being typed	3,20
LINE FEED	carriage goes to next line without terminating current line	3
BREAK	interrupts execution	3,30
\	deletes last character typed	20
*CTRL-W	deletes last word typed	20
*CTRL-N	only valid in EDIT LINE mode. Gets next word of line being edited	22
*CTRL-R	only valid in EDIT LINE mode. Gets rest of line being edited	22
*CTRL-B	same as /BLANK/	
*CTRL-G	same as /BELL/	
*CTRL-L	same as /FORM FEED/	

*this notation indicates that the CTRL key is pressed simultaneously with indicated letter.

Special Characters

	Description	Page
"	used to delimit literals	4
/	used to delimit names	5,13
( )	"noise" word, used for clarity	6,11
;	used to delimit comments	40
←	typed by LOGO to indicate user control	3
>	typed by LOGO to indicate define mode	5,22
#	typed by LOGO at beginning of line to indicate line has been "RUBbed OUT"	20
::	typed by LOGO to indicate REQUEST needs user type-in	32

LOGO Reference Manual

Addendum No. 1

1. Slash (/) has been replaced by colon or dots (:) as the delimiter for LOGO names.

2. The command LOCAL now takes only a single input (instead of an arbitrary number as before).

3. The number sign (#) embedded in a literal TYPEs and PRINTs as a blank space. Thus,

```
←MAKE "A" "A###B"
←LIST ALL NAMES
:A: IS "A###B"
←PRINT :A:
A   B
```

4. The special actions :LINEFEED:, :BELL:, :CARRIAGE RETURN:, :FORMFEED:, and :SKIP: are now also available as the zero-input commands LINEFEED, BELL, RETURN, FORMFEED, and SKIP.

5. The multi-word commands IF TRUE, IF FALSE, GO TO LINE, RESET CLOCK, and the multi-word operations TOUCH LEFT, and TOUCH RIGHT have been changed to the single words IFTRUE, IFFALSE, GOTOLINE, RESETCLOCK, TOUCHLEFT, and TOUCHRIGHT.

6. A one-input command, TYPEIN, has been added. TYPEIN "A" has the same effect as MAKE "A" REQUEST.

```
←TYPEIN "SAM"
*I AM SAM
←PRINT :SAM:
I AM SAM
←
```

7. The character > has been replaced by the character @ to indicate readiness in define mode.

LOGO Reference Manual

Addendum No. 2

1. The prefix arithmetic operations SUM, DIFFERENCE, PRODUCT, QUOTIENT have been supplemented by their infix forms +, -, *, /. > is now the infix form of GREATERP and both infix (<) and prefix (LESSP) versions of the predicate strictly-less-than have also been implemented. EQUALP and the equivalent infix = are new two-input predicates which test numerical equality. (EQUALP OF 0 AND 00 is "TRUE" whereas IS OF 0 AND 00 is "FALSE".)

Expressions containing only prefix operations and commands are parsed from left to right as before (see LOGO manual). Infix operations have their usual meanings in expressions. For example, 2 + 3 * 4 gives 14 and 3 = 1 + 2 gives "TRUE". (Using a simple left-to-right parsing, this last expression would have been interpreted as (3 = 1) + 2, i.e., "FALSE" + 2 and thus would have resulted in an error message.) Parsing is straightforwardly described by giving different precedence ranking to different classes of operators, as in the following table.

*,/
+,-
=,>,<
all prefix operators

Thus, * and / have identical precedence which is higher than the precedence of + and -.

Now we can easily state the general LOGO parsing rule. Parsing is still strictly left-to-right except for the difficult case where there are two possible operators with which to associate an input. This is the case where an input is preceded by any (prefix or infix) operator and is followed by an infix operator. Symbolically: Left-Op Input Right-Op. (For example in PRINT 2 + 3 we want the input 2 to go with the operator +, not with the operator PRINT.) The general rule is: the input goes with the operator on its left (Left-Op) unless the precedence of that operator is less than the precedence of the infix operator on the right. The following examples show the use of the rule.

```
←PRINT 2 + 3
5
←PRINT 4 * 3 = 6 * 2
TRUE
←PRINT 2 + 3 * 4 / 6
4
←PRINT BOTH 3 < 4 AND 3 > 4
FALSE
```

```
←PRINT WORD 1 AND 1 * WORD 2 AND 2
122
←PRINT BUTLAST 13 * 4
5
←PRINT 4 * BUTLAST 13
4
```

Parentheses can still be used to enclose valid LOGO expressions (see Pp. 6 and 11, LOGO Manual). Now, however, they can change the order of parsing.

```
←PRINT (BUTLAST 13) * 4
4
←PRINT (WORD 1 AND 1) * (WORD 2 AND 2)
242
```

Finally, a + or - which is not preceded by an input is taken as a unary + or -.

```
←PRINT -----1
-1
```

There are some difficulties, though,

```
←PRINT SUM -1 -1
1 INPUT MISSING FOR SUM
```

The second - was taken as binary resulting in PRINT SUM -2. Our parsing rule was chosen to minimize such difficulties. One should, when possible, avoid the use of mixed prefix and infix expressions which are not transparent. When writing mixed expressions, the use of parentheses or noise words should be encouraged to increase transparency. Thus, the previous example could be correctly written as

```
←PRINT SUM -1 AND -1
-2
```

2. The one-input operation NOT outputs "TRUE" if the input is "FALSE" and "FALSE" if the input is "TRUE". Any other input gives an error message.

3. WORDS and SENTENCES are operations which concatenate any number of inputs to form a LOGO word or a LOGO sentence. These operations keep taking inputs until a right parenthesis occurs or the line ends. Their built-in abbreviations are WS and SS.

```
←PRINT SENTENCES "A" "B" "C" "D"
A B C D
←PRINT (WORDS 1 2 3) + (WORDS 3 2 1)
444
```

4. An additional form of conditional: IF ____ THEN ____ ELSE ____ has been implemented. The IF command is followed by an expression which must evaluate to "TRUE" or "FALSE". If it is "TRUE", then the command following THEN is executed. If it is "FALSE", the command following ELSE is executed. The word THEN is a noise word which may be omitted. The ELSE and subsequent command may also be omitted. In that case, "FALSE" results in a null action.

```
←IF BOTH 1 > 2 AND 2 < 1 THEN PRINT "HELP" ELSE PRINT "WHEW"
WHEW
←IF 2 + 2 = 4 PRINT SUM OF 1 AND 2
3

←TO REVERSE :WORD:
10 IF EMPTYP :WORD: OUTPUT :EMPTY: ELSE OUTPUT WORD OF
  (LAST :WORD:) (REVERSE BUTLAST :WORD:)
END
```